Pro AngularJS with Parse.com

Some Pro AngularJS readers have experienced problems getting Deployd to work. In these alternative chapters, I demonstrate how to use the hosted Parse.com service to create the backend for the SportsStore application. Parse.com offers a free tier of access for project development and an account can be created quickly and easily. I have no relationship with Parse.com and selected their service because it has a good reputation and offers a similar approach to Deployd. Parse.com was on my shortlist of backend platforms when I wrote the book, but I ended up going with Deployd because I was concerned about hosted offerings changing or disappearing and Deployd offered a local installation. Please accept my apologies if you have had problems with Deployd.

I hope you find these replacement chapters helpful, and I wish you every success in your AngularJS projects. Please contact me at `adam@adam-freeman.com` if you encounter problems with these chapters.

Adam Freeman, London
May 2014

CHAPTER 6

SportsStore: A Real Application

In the previous chapters, I built quick and simple AngularJS applications. Small and focused examples allow me to demonstrate specific AngularJS features, but they can lack context. To help overcome this problem, I am going to create a simple but realistic e-commerce application.

My application, called SportsStore, will follow the classic approach taken by online stores everywhere. I will create an online product catalog that customers can browse by category and page, a shopping cart where users can add and remove products, and a checkout where customers can enter their shipping details and place their orders. I will also create an administration area that includes create, read, update, and delete (CRUD) facilities for managing the catalog—and I will protect it so that only logged-in administrators can make changes.

My goal in this chapter and those that follow is to give you a sense of what real AngularJS development is like by creating as realistic an example as possible. I want to focus on AngularJS, of course, so I have simplified the integration with external systems, such as the data store, and have omitted others entirely, such as payment processing.

The SportsStore example is one that I use in a few of my books, not least because it demonstrates the ways in which different frameworks, languages, and development styles can be used to achieve the same result. You don't need to have read any of my other books to follow this chapter, but you will find the contrasts interesting if you already own my *Pro ASP.NET* and *Pro ASP.NET MVC* books.

The AngularJS features that I use in the SportsStore application are covered in depth in later chapters. Rather than duplicate everything here, I tell you just enough to make sense for the example application and refer you to other chapters for in-depth information. You can either read the SportsStore chapters end to end and get a sense of how AngularJS works or jump to and from the details chapter to get into the depth. Either way, don't expect to understand everything right away—AngularJS has a lot of moving parts and the SportsStore application is intended to show you how they fit together without diving too deeply into the details that I spend the rest of the book covering.

UNIT TESTING

One of the reasons that I use the SportsStore application in different books is because it makes it easy to introduce unit testing early. AngularJS provides some excellent support for unit testing, but I don't describe it until the final chapter in the book. That's because you really need to understand how AngularJS works

before you can write comprehensive unit tests, and I don't want to include all of the required information and then duplicate it throughout the rest of the book.

That's not to say that unit testing with AngularJS is difficult or that you need to be an expert in AngularJS to write a unit test. Rather, the features that make unit testing simple depend on some key concepts that I don't describe until Parts 2 and 3. You can skip ahead to Chapter 25 now if you want to get an early start on unit testing, but my advice is to read the book in sequence so that you understand the foundation on which the unit test features are built.

Getting Started

There is some basic preparation required before I start on the application. The instructions in the following sections install some optional AngularJS features to set up the server that will deliver the data.

Preparing the Data

The first step is to use Parse.com to create a new account and start building the back-end that will support the application. Go to `http://parse.com` and click the `Sign Up` button. The account is free and the basic no-cost level of service is sufficient for creating the SportsStore application.

Enter a name, email address and password and click the `Sign up` button. You will be prompted for a name for the application, as shown in Figure 6-1. Enter `SportsStore` and select your company type from the drop-down list. As the figure shows, I selected `Individual Developer`, which doesn't require any supplementary information.

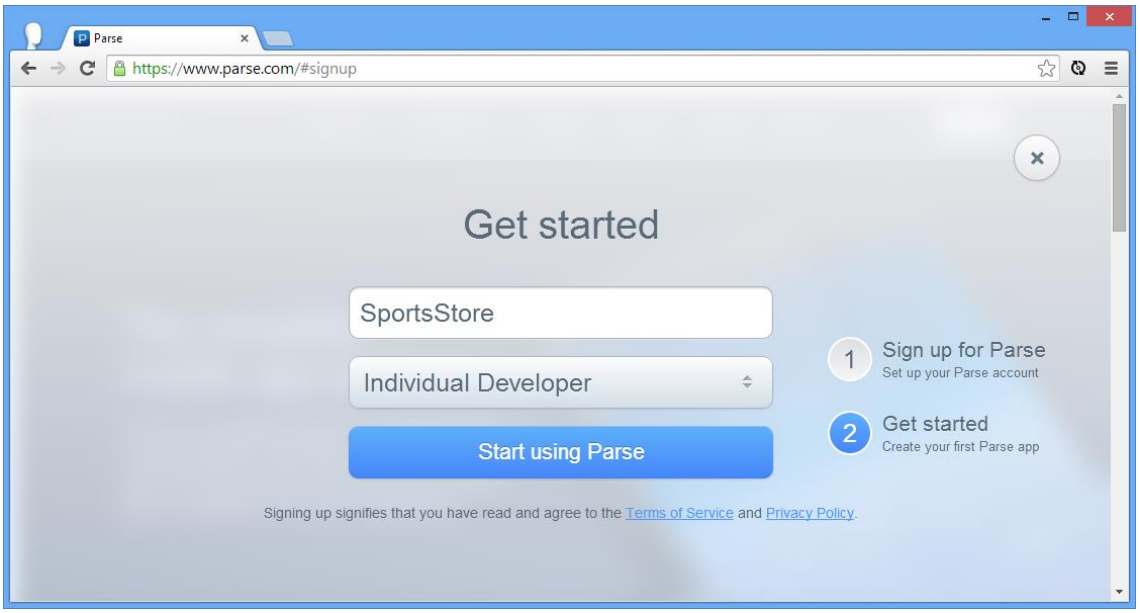

Figure 6-1. Registering with Parse.com

Once you have entered your details, click the `Start using Parse` button and you will be shown the welcome page for the `SportsStore` application backend. Click the `Data Browser` button to display the data view for the application.

Click on the `New Class` button, ensure that `Custom` is selected and enter `Products` as the name, as shown in Figure 6-2.

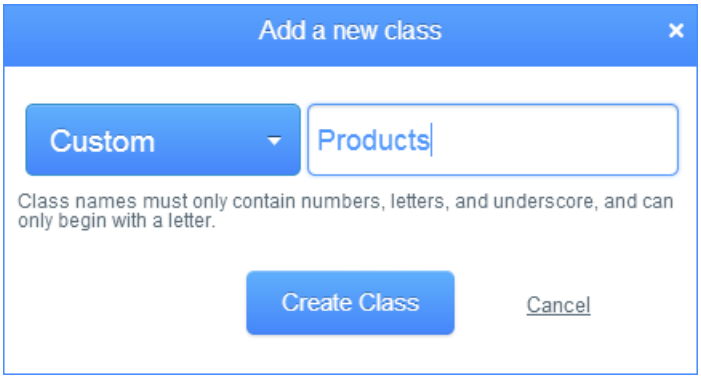

Figure 6-2. Creating a Parse.com New Class

Click the `Create Class` button and Parse.com will show a grid view where the application data will be displayed. The next step is to add columns to the grid that correspond to the data values that the SportsStore application will use.

Click the `+Col` button, select the `String` type and enter `name` into the text field, as shown in Figure 6-3.

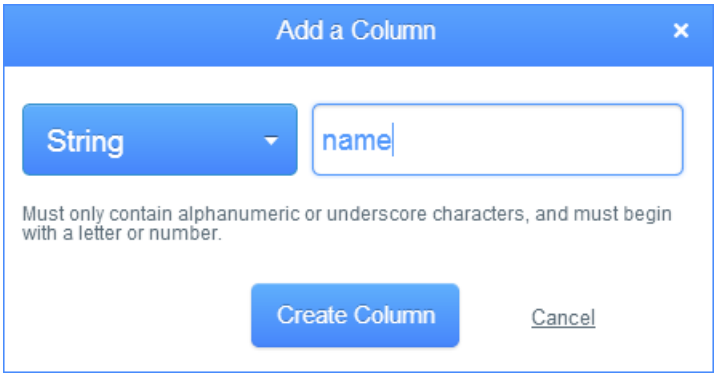

Figure 6-3. Creating a Product Column

Click the `Create Column` button and Parse.com will display the new column in the grid view, along with the default columns that Parse.com adds for all data types. Now repeat this process to create all of the columns shown in Table 6-1.

Table 6-1. *The Parse.com Columns Required for the Products Class*

Name	Type
name	String
description	String
category	String
price	number

Parse.com doesn't display the columns especially well, but if you scroll the page left-to-right, you will see that all of the columns have been added to the `Products` class.

Tip Notice that Parse.com has added an `objectId` property. This will be used to uniquely identify objects in the database. Parse.com will assign unique values to the `objectId` property automatically, and I'll be relying on these values when I implement the administration functions in Chapter 8.

Adding the Data

Now that I have defined the structure of the objects that Parse.com will store, I can add details of the products that the SportsStore will offer to customers. Click the `+Row` button and enter the details from the first row of Table 6-2 into the appropriate columns. Start with the `name` column and when you tab to the next column, Parse.com will automatically generate values for `ObjectId` and the other default columns. Repeat this process for all of the products listed in Table 6-2.

Table 6-2. *The Data for the Products Table*

Name	Description	Category	Price
Kayak	A boat for one person	Watersports	275
Lifejacket	Protective and fashionable	Watersports	48.95
Soccer Ball	FIFA-approved size and weight	Soccer	19.5
Corner Flags	Give your playing field a professional touch	Soccer	34.95
Stadium	Flat-packed 35,000-seat stadium	Soccer	79500.00
Thinking Cap	Improve your brain efficiency by 75%	Chess	16
Unsteady Chair	Secretly give your opponent a disadvantage	Chess	29.95
Human Chess Board	A fun game for the family	Chess	75

Bling-Bling King	Gold-plated, diamond-studded King	Chess	1200

When you have finished entering the data, the Parse.com Data Browser should look like Figure 6-4. (I edited this figure to emphasize the data that has to be entered and removed the default columns).

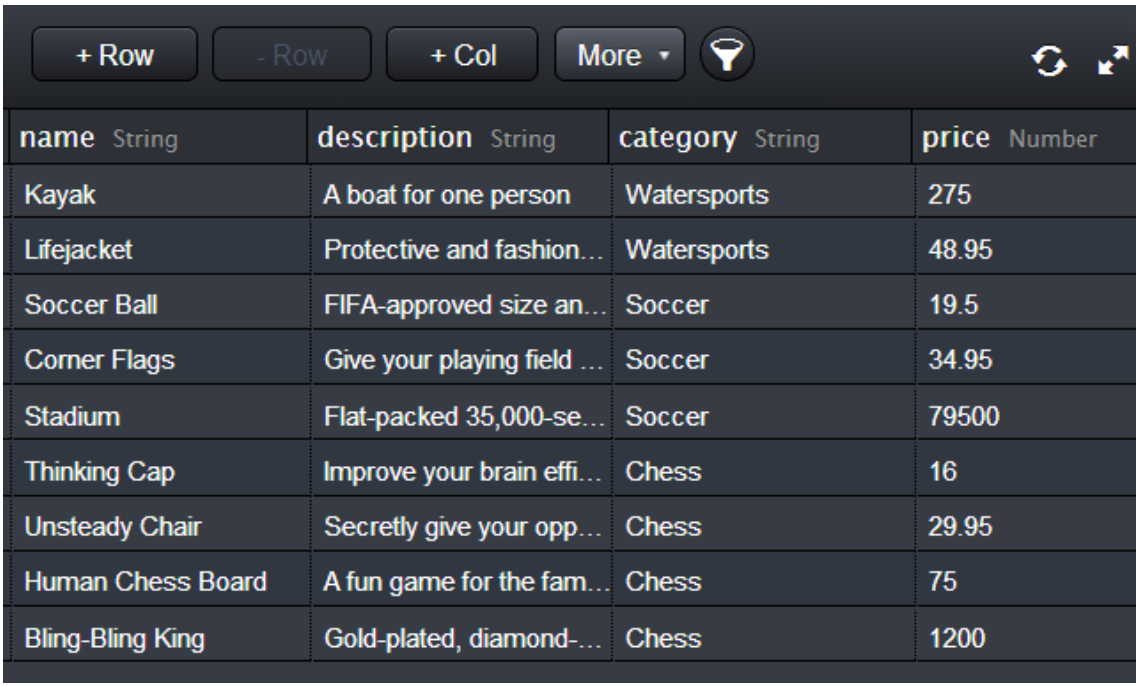

Figure 6-4. Entering the product data into the SportsStore dashboard

Tip I switched to the Dark layout using the slider at the bottom left of the page. The default color scheme gives better contrast for screen shots.

Getting the Application Keys

Parse.com requires clients to send keys as headers when making requests. To get the keys for your application, click the `Settings` button and then click the `Application Keys` button in the list at the left side of the window. The keys required for the AngularJS SportsStore application and the Application ID and the REST API Key. Make careful note of these because without them, your application will not function. Table 6-3 shows the keys that I will be using (but I will change them when these replacement chapters are published).

Tip The JavaScript Key is only used to test the Parse.com service in the next section.

Table 6-3. *The Parse.com Keys Required for the AngularJS SportsStore Application*

Key	Value
Application ID	`McOi2Q6FZNovOr4bQD6QRFwzgw4iRHOy2JcDYuLq`
REST API Key	`h4dnGScEtCWMMjaWmCJYNBZrDCrnROZmsKvEwXLD`
JavaScript Key	`ZoFCT89wmx9iOxFecPVEHLwq5HeIecJWjBVPIAUN`

Testing the Parse.com Data Service

To test that Parse.com is correctly configured and working, open a browser window and navigate to the a URL in the following format:

```
https://<appid>:javascript-key=<jskey>@api.parse.com/1/classes/Products
```

where `<appid>` is the value of the Application ID from Table 6-3 and `<jskey>` is the value of the JavaScript key.

Caution The class name in the Parse.com URL is case-sensitive. Ensure that your URL includes `Products` and not `products`.

Using the keys shown in Table 6-3, I would request the following URL to test my Parse.com backend service:

```
https://McOi2Q6FZNovOr4bQD6QRFwzgw4iRHOy2JcDYuLq:javascript-
key=ZoFCT89wmx9iOxFecPVEHLwq5HeIecJWjBVPIAUN@api.parse.com/1/classes/Products/
```

Don't worry about the nasty URL format because the key values will be set using HTTP headers in the SportsStore application.

Caution Notice that I have used `https` as the protocol in the test URL. Parse.com, like most hosted API platforms, won't accept connections unless they are made using SSL.

The `/1/classes/Products` URL is interpreted by Parse.com as a request for all of the product data, expressed as a JSON string. Some browsers, such as Google Chrome, will display the JSON response directly in the browser window, but others, such as Microsoft Internet Explorer, require you to download the JSON to a file. Either way, you should see the following data, which I have formatted to help clarity, although the value of the default Parse.com properties, including `objectId` will be different:

```
{"results":[
 {"category":"Watersports","description":"A boat for one person","name":"Kayak",
  "price":275,"createdAt":"2014-04-25T19:39:34.387Z",
  "updatedAt":"2014-04-25T19:42:54.644Z","objectId":"iteixu2Sn9"},
 {"category":"Chess","description":"Improve your brain efficiency by 75%",
  "name":"Thinking Cap","price":16,"createdAt":"2014-04-25T19:41:01.245Z",
  "updatedAt":"2014-04-25T19:43:17.688Z","objectId":"VqJRDzQuoz"},
 {"category":"Chess","description":"A fun game for the family",
  "name":"Human Chess Board","price":75,
  "createdAt":"2014-04-25T19:41:16.841Z","updatedAt":"2014-04-25T19:43:23.729Z",
  "objectId":"fxg7cUQMSn"},
 {"category":"Chess","description":"Gold-plated, diamond-studded King",
  "name":"Bling-Bling King","price":1200, "createdAt":"2014-04-25T19:41:25.036Z",
  "updatedAt":"2014-04-25T19:43:25.562Z","objectId":"r3f6yw6XCz"},
 {"category":"Watersports","description":"Protective and fashionable",
  "name":"Lifejacket","price":48.95,"createdAt":"2014-04-25T19:40:08.548Z",
  "updatedAt":"2014-04-25T19:42:59.741Z","objectId":"OeMdG86DMu"},
 {"category":"Soccer","description":"FIFA-approved size and weight",
   "name":"Soccer Ball","price":19.5,"createdAt":"2014-04-25T19:40:37.757Z",
   "updatedAt":"2014-04-25T19:43:02.694Z","objectId":"N2qLCwAfI2"},
 {"category":"Soccer","name":"Corner Flags","price":34.95,
   "description":"Give your playing field a professional touch",
   "createdAt":"2014-04-25T19:40:48.451Z",
   "updatedAt":"2014-04-25T19:43:09.906Z","objectId":"VauZGeizXR"},
 {"category":"Soccer",
   "description":"Flat-packed 35,000-seat stadium",
   "name":"Stadium","price":79500, "objectId":"m9uc1oV9EX",
   "createdAt":"2014-04-25T19:40:54.911Z","updatedAt":"2014-04-25T19:43:15.352Z"},
  {"category":"Chess", "name":"Unsteady Chair","price":29.95,
   "description":"Secretly give your opponent a disadvantage",
   "createdAt":"2014-04-25T19:41:10.141Z",
   "updatedAt":"2014-04-25T19:43:21.098Z","objectId":"hDdYYN4abc"}
]}
```

Preparing the Application

Before I start writing the application, I need to prepare the `angularjs` folder by creating a directory structure for the files that will make up the application and downloading the AngularJS and Bootstrap files that I will need.

Creating the Directory Structure

You can arrange the files that make up an AngularJS application in any way you like. You can even use predefined templates with some client-side development tools, but I am going to keep things simple and follow the basic layout that I use for most AngularJS projects. This isn't always the layout that I finish with, because I tend to move and regroup files as a project grows in complexity, but this is where I usually start. Create the directories described in Table 6-4 within the `angularjs` folder.

Table 6-4. *The Folders Required for the SportsStore Application*

Name	Description
`components`	Contains self-contained custom AngularJS components.
`controllers`	Contains the application's controllers. I describe controllers in Chapter 13.
`filters`	Contains custom *filters*. I describe filters in depth in Chapter 14.
`ngmodules`	Contains optional AngularJS modules. I describe the optional modules throughout this book and will give references for each of them as I apply them to the SportsStore application.
`views`	Contains the partial views for the SportsStore application. Views contain a mix of directives and filters, which I described in Chapters 10–17.

Installing the AngularJS and Bootstrap Files

My preference, without any real foundation in reason, is to put the main AngularJS JavaScript file and the Bootstrap CSS files into the main `angularjs` directory and put the optional AngularJS modules that I use into the `ngmodules` folder. I can't explain why I do this, but it has become a habit. Following the instructions in Chapter 1, copy the files I listed in Table 6-5 into the `angularjs` folder.

Table 6-5. *The Files to Be Installed in the angularjs Folder*

Name	Description
`angular.js`	The main AngularJS functionality
`bootstrap.css`	The Bootstrap CSS styles

`bootstrap-theme.css`	The default theme for the Bootstrap CSS files

Not all AngularJS functionality comes in the `angular.js` file. For the SportsStore application I will require some additional features that are available in optional modules. These are the files that I keep in the `ngmodules` folder. Following the instructions in Chapter 1, download the files described in Table 6-6 and place them in the `angularjs/ngmodules` folder.

Table 6-6. *The Optional Module Files to Be Installed in the ngmodules Folder*

Name	Description
`angular-route.js`	Adds support for URL routing. See Chapter 7 for URL routing in the SportsStore application, and see Chapter 22 for full details of this module.
`angular-resource.js`	Adds support for working with RESTful APIs. See Chapter 8 for REST in the SportsStore application, and see Chapter 21 for full details of this module.

Building the Basic Outline

I like to start a new AngularJS application by mocking up the basic structure with placeholder content and then filling in each part in turn. The basic layout of the SportsStore application is the classic two-column layout that you will find in many web stores—a set of categories in the first column that is used to filter the set of products displayed in the second column. Figure 6-5 shows the effect I am aiming for.

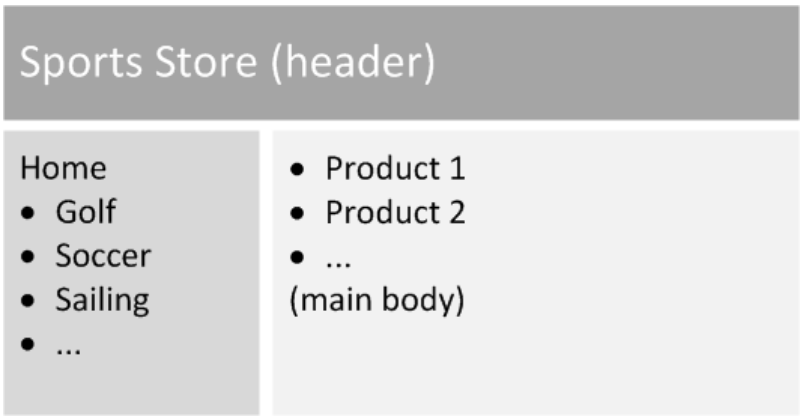

Figure 6-5. The two-column SportsStore layout

I'll add some additional features as I build the application, but the figure shows the initial functionality I will create. The first step is to create the top-level HTML file that will contain the structural markup and the `script` and `link` elements for the JavaScript and CSS files I will be using. Listing 6-1 shows the contents of the `app.html` file, which I created in the `angularjs` folder.

Listing 6-1. The Contents of the app.html File

```
<!DOCTYPE html>
<html ng-app="sportsStore">
<head>
    <title>SportsStore</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStore", []);
    </script>
</head>
<body>
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>
    <div class="panel panel-default row">
        <div class="col-xs-3">
            Categories go here
        </div>
        <div class="col-xs-8">
            Products go here
        </div>
    </div>
</body>
</html>
```

This file contains HTML elements that define the basic layout, styled using Bootstrap into a table structure, as described in Chapter 4. There are two AngularJS-specific aspects to this file. The first is the `script` element in which I call the `angular.module` method, as follows:

```
...
<script>
    angular.module("sportsStore", []);
</script>
...
```

Modules are the top-level building block in an AngularJS application, and this method call creates a new module called `sportsStore`. I don't do anything with the module other than create it at the moment, but I'll be using it to define functionality for the application later.

The second aspect is that I have applied the `ng-app` directive to the `html` element, like this:

```
...
<html ng-app="sportsStore">
...
```

The `ng-app` directive makes the functionality defined within the `sportsStore` module available within the HTML. I like to apply the `ng-app` directive to the `html` element, but you can be more specific, and a common alternative is to apply it to the `body` element instead.

Despite creating and applying an AngularJS module, the contents `app.html` file are simple and merely lay out the basic structure of the application, styled using Bootstrap. You can see how the browser displays the `app.html` file in Figure 6-6.

Tip To request the `app.html` file, I asked the browser to display the URL `http://localhost:5000/app.html`. I am using the Node.js web server that I introduced in Chapter 1, running on port 5000 of my local machine..

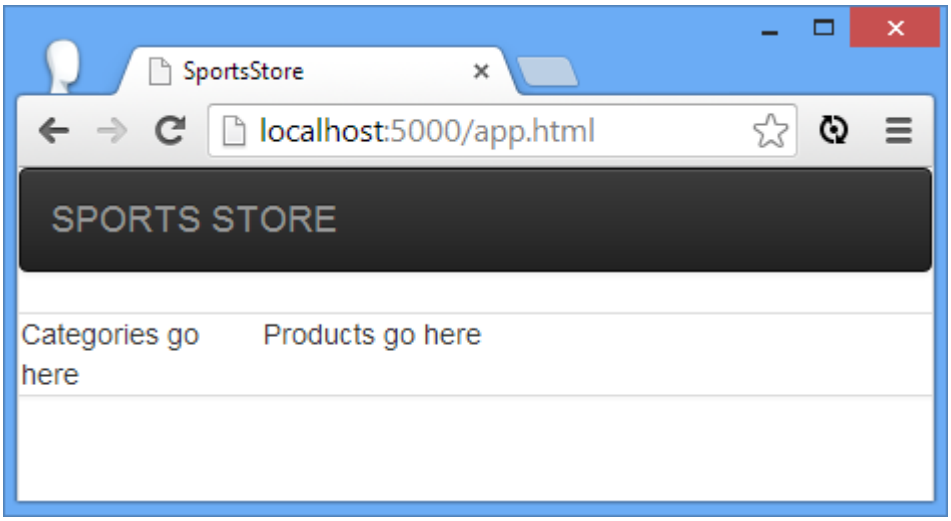

Figure 6-6. The initial layout of the SportsStore application

It doesn't look like much at the moment, but the application will start to take shape pretty quickly once the plumbing is in place and I start using AngularJS to build the application functionality.

Displaying the (Fake) Product Data

I am going to start by adding support for displaying the product data. I want to focus on one area of functionality at a time, so I am going to define fake local data initially, which I will then replace with data obtained from Parse.com in Chapter 7.

Creating the Controller

I need to start with a controller, which, as I explained in Chapter 3, defines the logic and data required to support a view on its scope. The controller I am going to create will be used throughout the application—something I refer to as the *top-level controller*, although this is a term of my own invention—and I define this controller in its own file. Later, I'll start to group multiple related controllers in a file, but I put the top-level controller in its own file. Listing 6-2 shows the contents of the `controllers/sportsStore.js` file, which I created for this purpose.

Tip The reason I keep the top-level controller in a separate file is so that I can keep an eye on it when it changes in a revision control system. The top-level controller tends to change a lot during the early stages of development, when the application is taking shape, and I don't want the avalanche of change notifications to mask when other controllers are being altered. Later in the project, when the main functionality is complete, the top-level controller changes infrequently, but when it does change, there is a potential for breaking pretty much everything else in the application. At that point in the development cycle, I want to know when someone alters the top-level controller so that I can ensure that the changes have been thought through and fully tested.

Listing 6-2. The Contents of the sportsStore.js File

```
angular.module("sportsStore")
.controller("sportsStoreCtrl", function ($scope) {

    $scope.data = {
        products: [
            { name: "Product #1", description: "A product",
                category: "Category #1", price: 100 },
            { name: "Product #2", description: "A product",
                category: "Category #1", price: 110 },
            { name: "Product #3", description: "A product",
                category: "Category #2", price: 210 },
            { name: "Product #4", description: "A product",
                category: "Category #3", price: 202 }]
    };
});
```

Notice that the first statement in this file is a call to the `angular.module` method. This is the same method call that I made in the `app.html` file to define the main module for the SportsStore application. The difference is that when I defined the module, I provided an additional argument, like this:

```
...
angular.module("sportsStore", []);
...
```

The second argument is an array, which is currently empty, that lists the modules on which the `sportsStore` module depends and tells AngularJS to locate and provide the functionality that these modules contain. I'll be adding elements to this array later, but for now it is important to know that when you supply the array—empty or otherwise—you are telling AngularJS to *create* a new module. AngularJS will report an error if you try to create a module that already exists, so you need to make sure your module names are unique.

By contrast, the call to the `angular.module` method in the `sportsStore.js` file doesn't have the second argument:

```
...
angular.module("sportsStore")
...
```

Omitting the second argument tells AngularJS that you want to locate a module that has already been defined. In this situation, AngularJS will report an error if the module specified doesn't exist, so you need to make sure the module has already been created.

Both uses of the `angular.module` method return a `Module` object that can be used to define application functionality. I have used the `controller` method that, as its name suggests, defines a controller, but I describe the full set of methods available—and the components they create—in Chapters 9 and 18. You will also see me use some of these methods as I build the SportsStore application.

Note I wouldn't usually put the call to create the main application module in the HTML file like this because it is simpler to put everything in the JavaScript file. The reason I split up the statements is because the dual uses of the `angular.module` method cause endless confusion and I wanted to draw your attention to it, even if that means putting a JavaScript statement in the HTML file that could be omitted.

The main role of the top-level controller in the SportsStore application is to define the data that will be used in the different views that the application will display. As you will see—and as I describe in detail in Chapter 13—an AngularJS can have multiple controllers arranged in a hierarchy. Controllers arranged in this way can inherit data and logic from controllers above them, and by defining the data in the top-level controller, I can make it easily available to the controllers that I will be defining later.

The data I have defined is an array of objects that have the same properties as the data that is stored by Parse.com, which allows me to get started before I start making Ajax requests to get the real product information.

Caution Notice that when I define the data on the controller's scope, I define the data objects in an array that I assign to a property called `products` on an object called `data`, which in turn is attached to the scope. You have to be careful when you define data you want to be inherited because if you assign properties directly to the scope (that is, `$scope.products = [data]`) because other controllers can read, but not always modify, the data. I explain this in detail in Chapter 13.

Displaying the Product Details

To display details of the products, I need to add some HTML markup to the `app.html` file. AngularJS makes it easy to display data, as Listing 6-3 shows.

Listing 6-3. Displaying Product Details in the app.html File

```
<!DOCTYPE html>
<html ng-app="sportsStore">
<head>
    <title>SportsStore</title>
```

```
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStore", []);
    </script>
    <script src="controllers/sportsStore.js"></script>
</head>
<body ng-controller="sportsStoreCtrl">
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>
    <div class="panel panel-default row">
        <div class="col-xs-3">
            Categories go here
        </div>
        <div class="col-xs-8">
            <div class="well" ng-repeat="item in data.products">
                <h3>
                    <strong>{{item.name}}</strong>
                    <span class="pull-right label label-primary">
                        {{item.price | currency}}
                    </span>
                </h3>
                <span class="lead">{{item.description}}</span>
            </div>
        </div>
    </div>
</body>
</html>
```

There are three different kinds of changes highlighted in this listing. The first is that I have added a `script` element that imports the `sportsStore.js` file from the `controllers` folder. This is the file that contains the `sportsStoreCtrl` controller. Because I defined the `sportsStore` module in the `app.html` file and then located and used it in the `sportsStore.js` file, I need to make sure the inline `script` element (the one that *defines* the module) appears before the one that imports the file (which *extends* the module).

The next change is to apply the controller to its view using the `ng-controller` directive, like this:

```
...
<body ng-controller="sportsStoreCtrl">
...
```

I will be using the `sportsStoreCtrl` controller to support the entire application, so I have applied it to the `body` element so that the view it supports is the entire set of content elements. This will start to make more sense when I begin to add other controllers to support specific features.

Generating the Content Elements

The last set of changes in Listing 6-3 creates the elements to display details of the products for sale in the SportsStore. One of the most useful directives that AngularJS provides is `ng-repeat`,

which generates elements for each object in an array of data. The ng-repeat directive is applied as an attribute whose value creates a local variable that is used for each data object in a specified array, like this:

```
...
<div class="well" ng-repeat="item in data.products">
...
```

The value I have used tells the ng-repeat directive to enumerate the objects in the data.products array applied to the scope by the controller for the view and assign each object to a variable called item. I can then refer to the current object in *data binding* expressions, which are denoted with the {{ and }} characters, like this:

```
...
<div class="well" ng-repeat="item in data.products">
    <h3>
        <strong>{{item.name}}</strong>
        <span class="pull-right label label-primary">{{item.price | currency}}</span>
    </h3>
    <span class="lead">{{item.description}}</span>
</div>
...
```

The ng-repeat directive duplicates the element to which it is applied (and any descendant elements) for each data object. That data object is assigned to the variable item, which allows me to insert the values of the name, price, and description properties as required.

The name and description values are inserted as-is in the HTML elements, but I have done something different with the price property: I have applied a *filter*. A filter formats or orders data values for display in a view. AngularJS comes with some built-in filters, including the currency filter, which formats numeric values as currency amounts. Filters are applied by using the | character, followed by the name of the filter, such that the expression item.price | currency tells AngularJS to pass the value of the price property of the item object through the currency filter.

The currency filter formats amounts as U.S. dollars by default, but, as I explain in Chapter 14, you can use some AngularJS localization filters to display other currency formats. I describe the built-in filters and show you how to create your own in Chapter 14. I will also create a custom filter in the next section. The result is that a set of elements like this one is generated for each element:

```
<div class="well ng-scope" ng-repeat="item in data.products">
    <h3>
        <strong class="ng-binding">Product #1</strong>
        <span class="pull-right label label-primary ng-binding">$100.00</span>
    </h3>
    <span class="lead ng-binding">A product</span>
</div>
```

Notice now AngularJS has annotated the elements with classes that begin with ng-. These are an artifact of AngularJS processing the elements and resolving data bindings, and you should not attempt to change them. You can see the visual effect changes in Listing 6-3 by

loading the `app.html` file in the browser, as shown in Figure 6-7. I have shown only the first couple of products, but all of the details are displayed in a single list (something I will address by adding pagination later in this chapter).

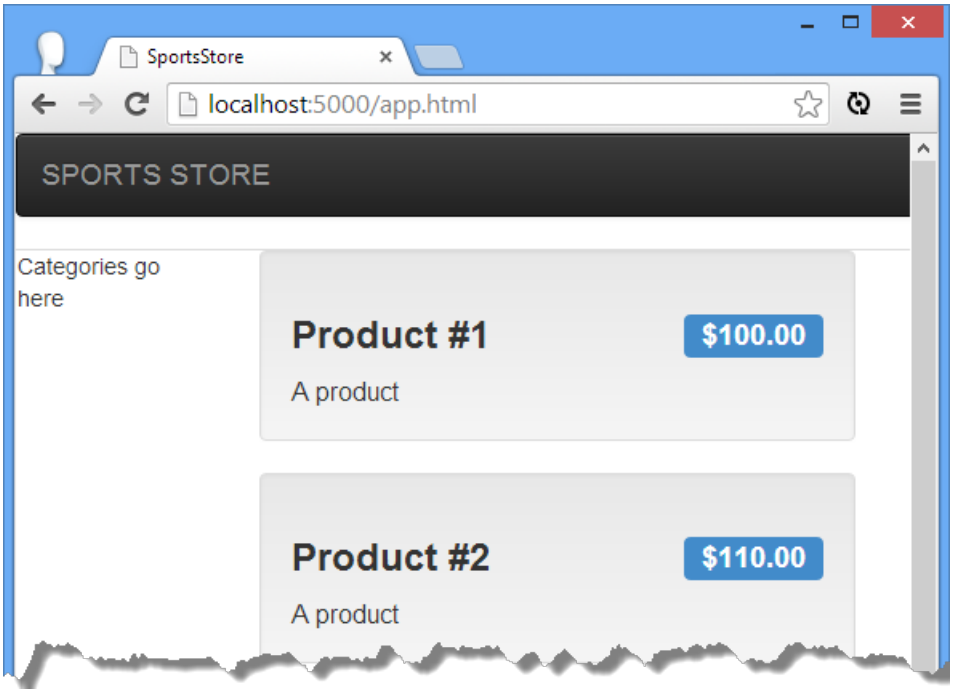

Figure 6-7. Generating the product detail elements

Displaying the Category List

The next step is to display the list of categories so that the user can filter the set of products that are displayed. Implementing this feature requires the generation of the elements with which the user will navigate, handling the navigation to select a product category, and, finally, updating the details pane so that only products in the selected category are displayed.

Creating a List of Categories

I want to generate the category elements dynamically from the product data objects, rather than hard-code HTML elements for a fixed set of categories. The dynamic approach is more complex to set up, but it will allow the SportsStore application to automatically reflect changes in the product catalog. This means I have to be able to generate a list of unique category names from an array of product data objects. This is a feature that AngularJS doesn't include but is easy to implement by creating and applying a custom filter. I created a file called `customFilters.js` in the `filters` directory, and you can see the contents of this file in Listing 6-4.

Listing 6-4. The Contents of the customFilters.js File

```
angular.module("customFilters", [])
    .filter("unique", function () {
        return function (data, propertyName) {
            if (angular.isArray(data) && angular.isString(propertyName)) {
                var results = [];
                var keys = {};
                for (var i = 0; i < data.length; i++) {
                    var val = data[i][propertyName];
                    if (angular.isUndefined(keys[val])) {
                        keys[val] = true;
                        results.push(val);
                    }
                }
                return results;
            } else {
                return data;
            }
        }
    });
```

Custom filters are created using the `filter` method defined by `Module` objects, which are obtained or created through the `angular.module` method. I have chosen to create a new module, called `customFilters`, to contain my filter, mainly so I can show you how to define and combine multiple modules within an application.

Tip There are no hard-and-fast rules about when you should add a component to an existing module or create a new one. I tend to create modules when I am defining functionality that I expect to reuse in a different application later. Custom filters tend to be reusable because data formatting is something that almost all AngularJS applications require and most developers end up with a utility belt of common formats they require.

The arguments to the `filter` method are the name of the filter, which is `unique` in this case, and a *factory function* that returns a *filter function* that does the actual work. AngularJS calls the factory function when it needs to create an instance of the filter, and the filter function is invoked to perform the filtering.

All filter functions are passed the data they are being asked to format, but my filter defines an additional argument, called `propertyName`, which I use to specify the object property that will be used to generate a list of unique values. You'll see how to specify the value for the `propertyName` argument when I apply the filter. The implementation of the filter function is simple: I enumerate the contents of the data array and build up a list of the unique values of the property whose name is provided through the `propertyName` argument.

Tip I could have hard-coded the filter function to look for the `category` property, but that limits the potential for reusing the `unique` filter elsewhere in the application or even in another AngularJS application. By taking the

name of the property as an argument, I have created a filter that can be used to generate a list of the unique values of any property in a collection of data objects.

A filter function is responsible for returning the filtered data, even if it is unable to process the data it receives. To that end, I check to see that the data I am working with is an array and that the `propertyName` is a string—checks that I perform using the `angular.isArray` and `angular.isString` methods. Later in the code, I check to see whether a property has been defined using the `angular.isUndefined` method. AngularJS provides a range of useful utility methods, including ones that allow you to check the type of objects and properties. I describe the complete set of these methods in Chapter 5. If my filter has received an array and a property name, then I generate and return an array of the unique property values. Otherwise, I return the data I have received unmodified.

Tip Changes that a filter makes to the data affect only the content displayed to the user and do not modify the original data in the scope.

Generating the Category Navigation Links

The next step is to generate the links that the user will click to navigate between product categories. This requires the use of the `unique` filter that I created in the previous section, along with some useful built-in AngularJS features, as shown in Listing 6-5.

Listing 6-5. Generating the Navigation Links in the app.html File

```
<!DOCTYPE html>
<html ng-app="sportsStore">
<head>
    <title>SportsStore</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStore", ["customFilters"]);
    </script>
    <script src="controllers/sportsStore.js"></script>
    <script src="filters/customFilters.js"></script>
</head>
<body ng-controller="sportsStoreCtrl">
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>
    <div class="panel panel-default row">
        <div class="col-xs-3">
            <a ng-click="selectCategory()"
               class="btn btn-block btn-default btn-lg">Home</a>
            <a ng-repeat="item in data.products | orderBy:'category' | unique:'category'"
```

```
                ng-click="selectCategory(item)" class=" btn btn-block btn-default btn-lg">
                 {{item}}
            </a>
        </div>
        <div class="col-xs-8">
            <div class="well" ng-repeat="item in data.products">
                <h3>
                    <strong>{{item.name}}</strong>
                    <span class="pull-right label label-primary">
                        {{item.price | currency}}
                    </span>
                </h3>
                <span class="lead">{{item.description}}</span>
            </div>
        </div>
    </div>
</body>
</html>
```

The first change that I made in this listing was to update the definition of the `sportsStore` module to declare a dependency on the `customFilters` module that I created in Listing 6-5 and that contains the `unique` filter:

```
...
angular.module("sportsStore", ["customFilters"]);
...
```

This is known as *declaring a dependency*. In this case, I am declaring that the `sportsStore` module depends on the functionality in the `customFilters` module. This causes AngularJS to locate the `customFilters` module and make it available so that I can refer to the components it contains, such as filters and controllers—a process known as *resolving the dependency*.

Tip The process of declaring and managing dependencies between modules and other kinds of components—known as *dependency injection*—is central to AngularJS. I explain the process in Chapter 9.

I also have to add a `script` element that loads the contents of the file that contains the `customFilters` module, as follows:

```
...
<script>
    angular.module("sportsStore", ["customFilters"]);
</script>
<script src="controllers/sportsStore.js"></script>
<script src="filters/customFilters.js"></script>
...
```

Notice that I am able to define the `script` element for the `customFilters.js` file *after* the one that creates the `sportsStore` module and declares a dependency on the `customFilters` module. This is because AngularJS loads all of the modules before using them to resolve dependencies. The effect can be confusing: The order of the `script` elements is important when you are *extending* a module (because the module must already have been defined) but not when *defining*

a new module or declaring a dependency on one. The final set of changes in Listing 6-5 generates the category selection elements. There is quite a lot going on in these elements, and it will be easier to understand if you know what the result looks like—the addition of the category buttons—shown in Figure 6-8.

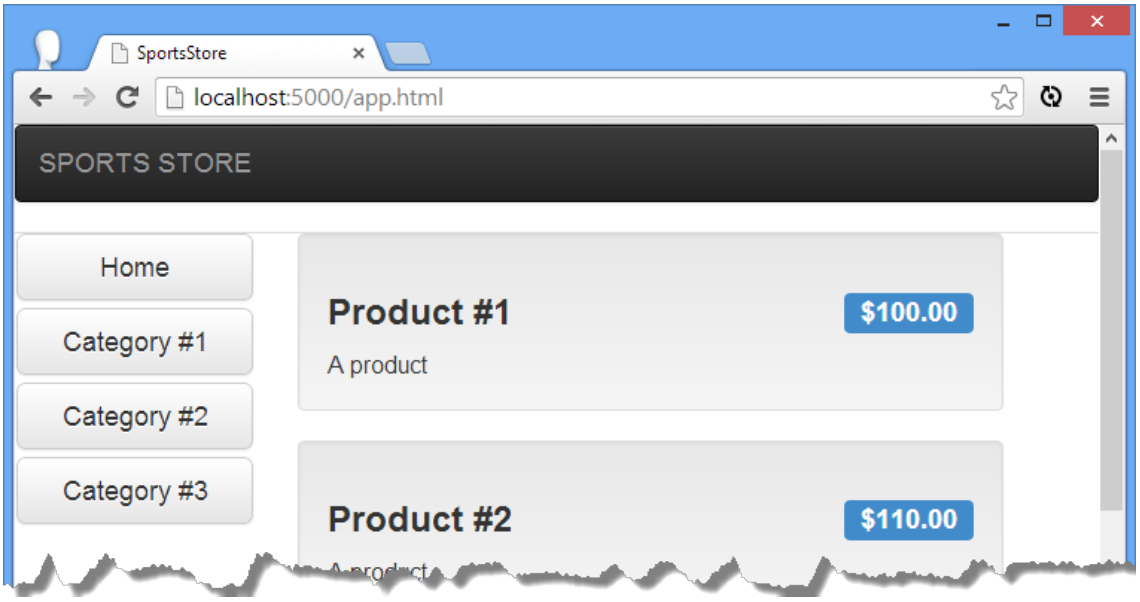

Figure 6-8. The category navigation buttons

Generating the Navigation Elements

The most interesting part of the markup is the use of the `ng-repeat` element to generate an `a` element for each product category, as follows:

```
...
<a ng-click="selectCategory()" class="btn btn-block btn-default btn-lg">Home</a>
<a ng-repeat="item in data.products | orderBy:'category' | unique:'category'"
    ng-click="selectCategory(item)" class=" btn btn-block btn-default btn-lg">
        {{item}}
</a>
...
```

The first part of the `ng-repeat` attribute value is the same as the one I used when generating the product details, `item in data.products`, and tells the `ng-repeat` directive that it should enumerate the objects in the `data.products` array, assign the current object to a variable called `item`, and duplicate the `a` element to which the directive has been applied.

The second part of the attribute value tells AngularJS to pass the `data.products` array to a built-in filter called `orderBy`, which is used to sort arrays. The `orderBy` filter takes an argument that specifies which property the objects will be sorted by, which I specify by placing a colon

(the : character) after the filter name and then the argument value. In this example, I have specified that the category property be used. (I describe the orderBy filter fully in Chapter 14.)

Tip Notice that I have specified the name of the property between single quotes (the ' character). By default, AngularJS assumes that names in expression refer to variables defined on the scope. To specify a static value, I have to use a string literal, which requires the single quote characters in JavaScript. (I could have used double quotes, but I already used them to demark the start and end of the ng-repeat directive attribute value.)

The use of the orderBy filter puts the product objects in order, sorted by the value of their category property. But one of the nice features of filters is that you can chain several together by using the bar symbol (the | character) and the name of another filter. In this case, I have used the unique filter that I developed earlier in the chapter. AngularJS applies filters in the order in which they are applied, which means that the objects are sorted by the category property and only then passed to the unique filter, which generates the set of unique category values. You can see how I have specified the property the unique filter will operate on:

```
...
<a ng-repeat="item in data.products | orderBy:'category' | unique:'category'"
...
```

The effect is that the data.products array is passed to the orderBy filter, which sorts the objects based on the value of the category property. The sorted array is then passed to the unique array, which returns a string array that contains the set of unique category values—and since the unique filter doesn't change the order of the values it processes, the results remain sorted by the previous filter.

Or, to put it more directly, this is an instruction to the ng-repeat directive to generate a set of unique category names, enumerate each of them, assign the current value to a variable called item, and generate an a element for each value.

Tip I could have reversed the filters and achieved the same effect. The difference would be that the orderBy filter would be operating on an array of strings, rather than product objects (because that's what the unique filter produces as its result). The orderBy filter is designed to operate on objects, but you can sort strings by using this incantation: orderBy:'toString()'. Don't forget the quotes; otherwise, AngularJS will look for a scope property called toString, rather than invoking the toString method.

Handling the Click Event

I used the ng-click directive on the a elements so that I can respond when the user clicks of the buttons. AngularJS provides a set of built-in directives, which I describe in Chapter 11, that make it easy to call controller behaviors in response to events. As its name suggests, the ng-click directive specifies what AngularJS should do when the click event is triggered, as follows:

```
...
<a ng-click="selectCategory()" class="btn btn-block btn-default btn-lg">Home</a>
<a ng-repeat="item in data.products | orderBy:'category' | unique:'category'"
    ng-click="selectCategory(item)" class=" btn btn-block btn-default btn-lg">
        {{item}}
</a>
...
```

There are two `a` elements in the `app.html` file. The first is static and creates the Home button, which I will use to display all of the products in all of the categories. For this element, I have set the `ng-click` directive so that it calls a controller behavior called `selectCategory` with no arguments. I'll create the behavior shortly, but for now, the important thing to note is that for the other `a` element—the one to which the `ng-repeat` directive has been applied—I have set up the `ng-click` directive so that it calls the `selectCategory` behavior with the value of the `item` variable as the argument. When the `ng-repeat` directive generates an `a` element for each unique category, the `ng-click` directive will be automatically configured such that the `selectCategory` behavior will be passed the category for the button, such as `selectCategory('Category #1')`, for example.

Selecting the Category

Clicking the category buttons in the browser doesn't have any effect at the moment because the `ng-click` directive on the `a` elements is set up to call a controller behavior that isn't defined. AngularJS doesn't complain when you try to access a nonexistent behavior or data value on the scope on the basis that it might be defined at some point in the future. This can make debugging a little frustrating because typos don't result in errors, but the flexibility that this approach gives is generally useful, as I explain in Chapter 13 when I describe how controllers and their scopes work in more depth.

Defining the Controller

I need to define a controller behavior called `selectCategory` in order to respond to the user clicking the category buttons. I don't want to add the behavior to the top-level `sportsStoreCtrl` controller, which I am reserving for behaviors and data that are required for the entire application. Instead, I am going to create a new controller that will be used just by the product listing and category views. Listing 6-6 shows the contents of the `controllers/productListControllers.js` file, which I added to the project in order to define the new controller.

Tip You may be wondering why I used a more specific name for the controller's file than for the one that contains filters. The reason is that filters are more generic and readily reused in other parts of the application or even other applications, whereas the kind of controller I am creating in this section tends to be tied to specific functionality. (This isn't true for all controllers, however, as you'll see in Chapters 15–17 when I show you how to create custom directives.)

Listing 6-6. The Contents of the productListControllers.js File

```
angular.module("sportsStore")
    .controller("productListCtrl", function ($scope, $filter) {

        var selectedCategory = null;

        $scope.selectCategory = function (newCategory) {
            selectedCategory = newCategory;
        }

        $scope.categoryFilterFn = function (product) {
            return selectedCategory == null ||
                product.category == selectedCategory;
        }
    });
```

I call the `controller` method on the `sportsStore` module that is defined in the `app.html` file (remember that one argument to the `angular.module` method means find an existing module, while two arguments means create a new one).

The controller is called `productListCtrl`, and it defines a behavior called `selectCategory`, matching the name of the behavior that the `ng-click` directives in Listing 6-6. The controller also defines `categoryFilterFn`, which takes a product object as its argument and returns `true` if no category has been selected or if a category has been selected and the product belongs to it—this will be useful shortly when I add the controller to the view.

Tip Notice that the `selectedCategory` variable is not defined on the scope. It is just a regular JavaScript variable, and that means it cannot be accessed from directives or data bindings in the view. The effect I have created is that the `selectCategory` behavior can be called to set the category, and the `categoryFilterFn` can be used to filter the product objects, but details of which category has been selected remains private. I won't be relying on this feature in the SportsStore applications—I just wanted to draw your attention to how controllers (and most other kinds of AngularJS components) can be selective about what public services and data they provide.

Applying the Controller and Filtering the Products

I have to apply the controller to the view using the `ng-controller` directive so that the `ng-click` directive is able to invoke the `selectCategory` behavior. Otherwise, the scope for the elements that contain the `ng-click` directive would be the one created by the top-level `sportsStoreCtrl` controller that doesn't contain the behavior. You can see the changes I have made to do this in Listing 6-7.

Listing 6-7. Applying a Controller in the app.html File

```
<!DOCTYPE html>
<html ng-app="sportsStore">
<head>
    <title>SportsStore</title>
```

```
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStore", ["customFilters"]);
    </script>
    <script src="controllers/sportsStore.js"></script>
    <script src="filters/customFilters.js"></script>
    <script src="controllers/productListControllers.js"></script>
</head>
<body ng-controller="sportsStoreCtrl">
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>
    <div class="panel panel-default row" ng-controller="productListCtrl">
        <div class="col-xs-3">
            <a ng-click="selectCategory()"
               class="btn btn-block btn-default btn-lg">Home</a>
            <a ng-repeat="item in data.products | orderBy:'category' | unique:'category'"
               ng-click="selectCategory(item)" class=" btn btn-block btn-default btn-lg">
                {{item}}
            </a>
        </div>
        <div class="col-xs-8">
            <div class="well"
                 ng-repeat="item in data.products | filter:categoryFilterFn">
                <h3>
                    <strong>{{item.name}}</strong>
                    <span class="pull-right label label-primary">
                        {{item.price | currency}}
                    </span>
                </h3>
                <span class="lead">{{item.description}}</span>
            </div>
        </div>
    </div>
</body>
</html>
```

I have added a `script` element to import the `productListControllers.js` file and applied the `ng-controller` directive for the `productListCtrl` controller on the part of the view that contains both the list of categories and the list of products.

Placing the `ng-controller` directive for the `productListCtrl` controller within the scope of the one for the `sportsStoreCtrl` controller means I can take advantage of *controller scope inheritance*, which I explain in detail in Chapter 13. The short version is the scope for the `productListCtrl` inherits the `data.products` array and any other data and behaviors that `sportsStoreCtrl` defines, which are then passed on to the view for the `productListCtrl` controller, along with any data or behaviors that it defines. The benefit of using this technique is that it allows you to limit the scope of controller functionality to the part of the application where it will be used, which makes it easier to perform good unit tests (as described in Chapter 25) and prevents unexpected dependencies between components in the application.

There is one other change in Listing 6-7: I changed the configuration of the `ng-repeat` directive that generates the product details, like this:

```
...
<div class="well" ng-repeat="item in data.products | filter:categoryFilterFn">
...
```

One of the built-in filters that AngularJS provides is called, confusingly, `filter`. It processes a collection and selects a subset of the objects it contains. I describe filters in Chapter 14, but the technique I am using here is to specify the name of the function defined by the `productListCtrl` controller. By applying the filter to the `ng-repeat` directive that creates the product details, I ensure that only the products in the currently selected category are displayed, as illustrated by Figure 6-9.

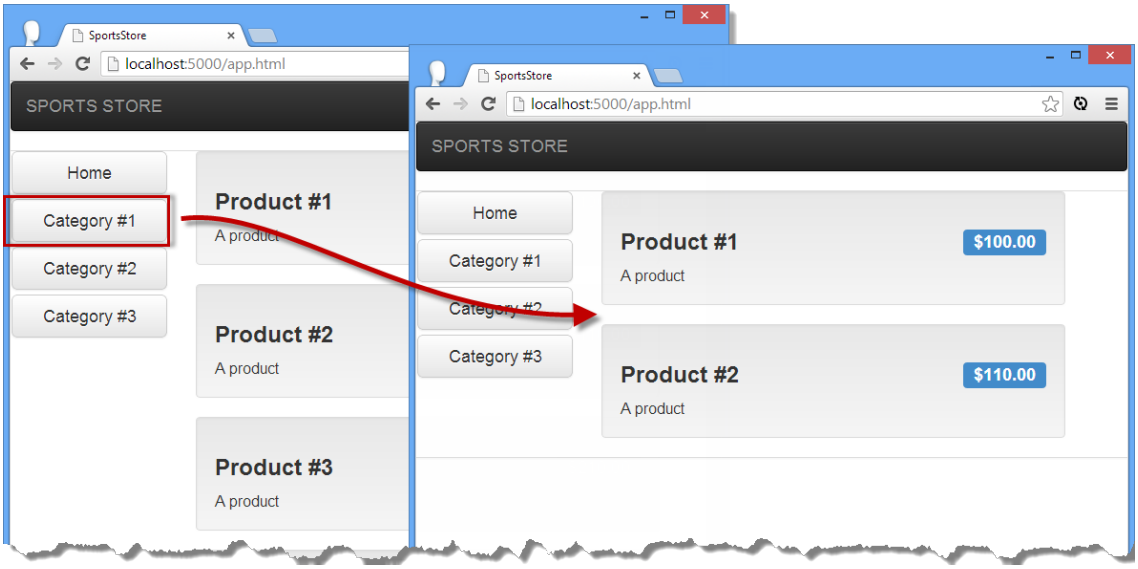

Figure 6-9. Selecting a category

Highlighting the Selected Category

The user can click the category buttons to filter the products, but there is no visual feedback to show which category has been selected. To address this, I am going to selectively apply the Bootstrap `btn-primary` CSS class to the category button that corresponds to the selected category. The first step is to add a behavior to the controller that will accept a category and, if it is the selected category, return the CSS class name, as shown in Listing 6-8.

Tip Notice how I am able to chain together method calls on an AngularJS module. This is because the methods defined by the `Module` return the `Module`, creating what is commonly referred to as a *fluent API*.

Listing 6-8. Returning the Bootstrap Class Name in the productListControllers.js File

```
angular.module("sportsStore")
```

```
    .constant("productListActiveClass", "btn-primary")
    .controller("productListCtrl", function ($scope, $filter, productListActiveClass) {

        var selectedCategory = null;

        $scope.selectCategory = function (newCategory) {
            selectedCategory = newCategory;
        }

        $scope.categoryFilterFn = function (product) {
            return selectedCategory == null ||
                product.category == selectedCategory;
        }

        $scope.getCategoryClass = function (category) {
            return selectedCategory == category ? productListActiveClass : "";
        }
    });
```

I don't want to embed the name of the class in the behavior code, so I have used the `constant` method on the `Module` object to define a fixed value called `productListActiveClass`. This will allow me to change the class that is used in one place and have the change take effect wherever it is used. To access the value in the controller, I have to declare the constant name as a dependency, like this:

```
...
.controller("productListCtrl", function ($scope, $filter, productListActiveClass) {
...
```

I can then use the `productListActiveClass` value in the `getCategoryClass` behavior, which simply checks the category it receives as an argument and returns either the class name or the empty string.

The `getCategoryClass` behavior may seem a little odd, but it is going to be called by each of the category navigation buttons, each of which will pass the name of the category it represents as the argument. To apply the CSS class, I use the `ng-class` directive, which I have applied to the `app.html` file in Listing 6-9.

Listing 6-9. Applying the ng-class Directive to the app.html File

```
...
<div class="col-xs-3">
    <a ng-click="selectCategory()"
        class="btn btn-block btn-default btn-lg">Home</a>
    <a ng-repeat="item in data.products | orderBy:'category' | unique:'category'"
        ng-click="selectCategory(item)" class=" btn btn-block btn-default btn-lg"
        ng-class="getCategoryClass(item)">
        {{item}}
    </a>
</div>
...
```

The `ng-class` attribute, which I describe in Chapter 11, will add the element to which it has been applied to the classes returned by the `getCategoryClass` behavior. You can see the effect this creates in Figure 6-10.

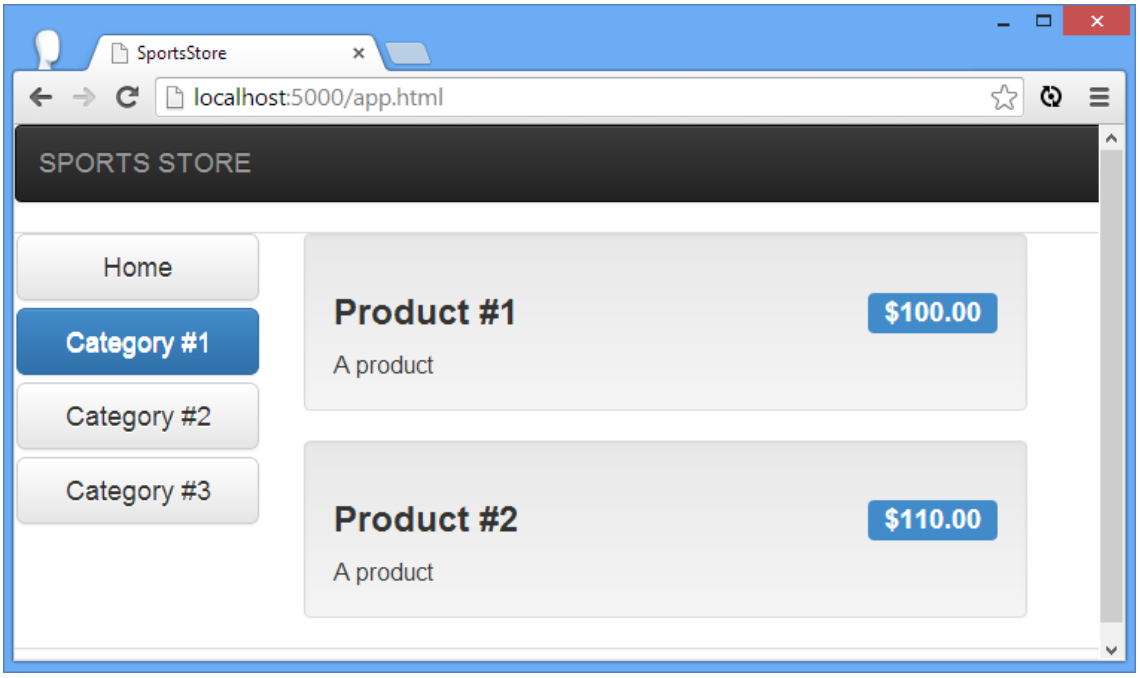

Figure 6-10. Highlighting the selected category

Adding Pagination

The last feature I am going to add in this chapter is *pagination*, such that only a certain number of product details are displayed at once. I don't really have enough data to make pagination terribly important, but it is a common requirement and worth demonstrating. There are three steps to implementing pagination: modify the controller so that the scope tracks the pagination state, implement filters, and update the view. I explain each step in the sections that follow.

Updating the Controller

I have updated the `productListCtrl` controller to support pagination, as shown in Listing 6-10.

Listing 6-10. Updating the Controller to Track Pagination in the productListControllers.js File

```
angular.module("sportsStore")
    .constant("productListActiveClass", "btn-primary")
    .constant("productListPageCount", 3)
    .controller("productListCtrl", function ($scope, $filter,
        productListActiveClass, productListPageCount) {

        var selectedCategory = null;

        $scope.selectedPage = 1;
        $scope.pageSize = productListPageCount;
```

```
        $scope.selectCategory = function (newCategory) {
            selectedCategory = newCategory;
            $scope.selectedPage = 1;
        }

        $scope.selectPage = function (newPage) {
            $scope.selectedPage = newPage;
        }

        $scope.categoryFilterFn = function (product) {
            return selectedCategory == null ||
                product.category == selectedCategory;
        }

        $scope.getCategoryClass = function (category) {
            return selectedCategory == category ? productListActiveClass : "";
        }

        $scope.getPageClass = function (page) {
            return $scope.selectedPage == page ? productListActiveClass : "";
        }
    });
```

The number of products shown on a page is defined as a constant called `productListPageCount`, which I have declared as a dependency of the controller. Within the controller I define variables on the scope that expose the constant value (so I can access it in the view) and the currently selected page. I have defined a behavior, `selectPage`, that allows the selected page to be changed and another, `getPageClass`, that is designed for use with the `ng-class` directive to highlight the selected page, much as I did with the selected category earlier.

Tip You might be wondering why the view can't access the constant values directly, instead of requiring everything to be explicitly exposed via the scope. The answer is that AngularJS tries to prevent tightly coupled components, which I described in Chapter 3. If views could access services and constant values directly, then it would be easy to end up with endless couplings and dependencies that are hard to test and hard to maintain.

Implementing the Filters

I have created two new filters to support pagination, both of which I have added to the `customFilters.js` file, as shown in Listing 6-11.

Listing 6-11. Adding Filters to the customFilters.js File

```
angular.module("customFilters", [])
.filter("unique", function () {
    return function (data, propertyName) {
        if (angular.isArray(data) && angular.isString(propertyName)) {
            var results = [];
            var keys = {};
            for (var i = 0; i < data.length; i++) {
```

```
                var val = data[i][propertyName];
                if (angular.isUndefined(keys[val])) {
                    keys[val] = true;
                    results.push(val);
                }
            }
            return results;
        } else {
            return data;
        }
    }
})
.filter("range", function ($filter) {
    return function (data, page, size) {
        if (angular.isArray(data) && angular.isNumber(page) && angular.isNumber(size)) {
            var start_index = (page - 1) * size;
            if (data.length < start_index) {
                return [];
            } else {
                return $filter("limitTo")(data.splice(start_index), size);
            }
        } else {
            return data;
        }
    }
})
.filter("pageCount", function () {
    return function (data, size) {
        if (angular.isArray(data)) {
            var result = [];
            for (var i = 0; i < Math.ceil(data.length / size) ; i++) {
                result.push(i);
            }
            return result;
        } else {
            return data;
        }
    }
});
```

The first new filter, called `range`, returns a range of elements from an array, corresponding to a page of products. The filter accepts arguments for the currently selected page (which is used to determine the start index of range) and the page size (which is used to determine the end index).

The `range` filter isn't especially interesting, other than I have built on the functionality provided by one of the built-in filters, called `limitTo`, which returns up to a specified number of items from an array. To use this filter, I have declared a dependency on the `$filter` service, which lets me create and use instances of filter. I explain how this works in detail in Chapter 14, but the key statement from the listing is this one:

```
...
return $filter("limitTo")(data.splice(start_index), size);
...
```

The result is that I use the standard JavaScript `splice` method to select part of the data array and then pass it to the `limitTo` filter to select no more than the number of items that can be

displayed on the page. The `limitTo` filter ensures that there are no problems stepping over the end of the array and will return fewer items if the specified number isn't available.

The second filter, `pageCount`, is a dirty—but convenient—hack. The `ng-repeat` directive makes it easy to generate content, but it works only on data arrays. You can't, for example, have it repeat a specified number of times. My filter works out how many pages an array can be displayed in and then creates an array with that many numeric values. So, for example, if a data array can be displayed in three pages, then the result from the `pageCount` filter would be an array containing the values `1`, `2`, and `3`. You'll see why this is useful in the next section.

Caution I am abusing the filter functionality to get around a limitation of the `ng-repeat` directive. This is a bad thing, but it is expedient and, as you will see, allows me to build on some of the functionality I created for related features. The better alternative would be to create a custom replacement for the `ng-repeat` directive that will generate elements a specified number of times. I explain the techniques required to do this—which are rather advanced—in Chapters 16 and 17.

Updating the View

The last step to implement pagination is to update the view so that only one page of products is displayed and to provide the user with buttons to move from one page to another. You can see the changes I have made to the `app.html` file in Listing 6-12.

Listing 6-12. Adding Pagination to the app.html File

```
<!DOCTYPE html>
<html ng-app="sportsStore">
<head>
    <title>SportsStore</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStore", ["customFilters"]);
    </script>
    <script src="controllers/sportsStore.js"></script>
    <script src="filters/customFilters.js"></script>
    <script src="controllers/productListControllers.js"></script>
</head>
<body ng-controller="sportsStoreCtrl">
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>
    <div class="panel panel-default row" ng-controller="productListCtrl">
        <div class="col-xs-3">
            <a ng-click="selectCategory()"
                class="btn btn-block btn-default btn-lg">Home</a>
            <a ng-repeat="item in data.products | orderBy:'category' | unique:'category'"
                ng-click="selectCategory(item)" class=" btn btn-block btn-default btn-lg"
                ng-class="getCategoryClass(item)">
```

```
                {{item}}
            </a>
        </div>
        <div class="col-xs-8">
            <div class="well"
                 ng-repeat=
         "item in data.products | filter:categoryFilterFn | range:selectedPage:pageSize">
                <h3>
                    <strong>{{item.name}}</strong>
                    <span class="pull-right label label-primary">
                        {{item.price | currency}}
                    </span>
                </h3>
                <span class="lead">{{item.description}}</span>
            </div>
            <div class="pull-right btn-group">
                <a ng-repeat=
                   "page in data.products | filter:categoryFilterFn | pageCount:pageSize"
                   ng-click="selectPage($index + 1)" class="btn btn-default"
                   ng-class="getPageClass($index + 1)">
                    {{$index + 1}}
                </a>
            </div>
        </div>
    </div>
</body>
</html>
```

The first change is to the `ng-repeat` directive that generates the product list so that the data is passed through the `range` filter to select the products for the current page. The details of the current page and the number of products per page are passed to the filter as arguments using the values I defined on the controller scope.

The second change is the addition of the page navigation buttons. I use the `ng-repeat` directive to work out how many pages the products in the currently selected category requires and pass the result to the `pageCount` filter, which then causes the `ng-repeat` directive to generate the right number of page navigation buttons. The currently selected page is indicated through the `ng-class` directive, and the page is changed through the `ng-click` directive.

You can see the result in Figure 6-11, which shows the two pages required to display all of the products. There are not enough items in the fake data for any one category to require multiple pages, but the effect is evident.

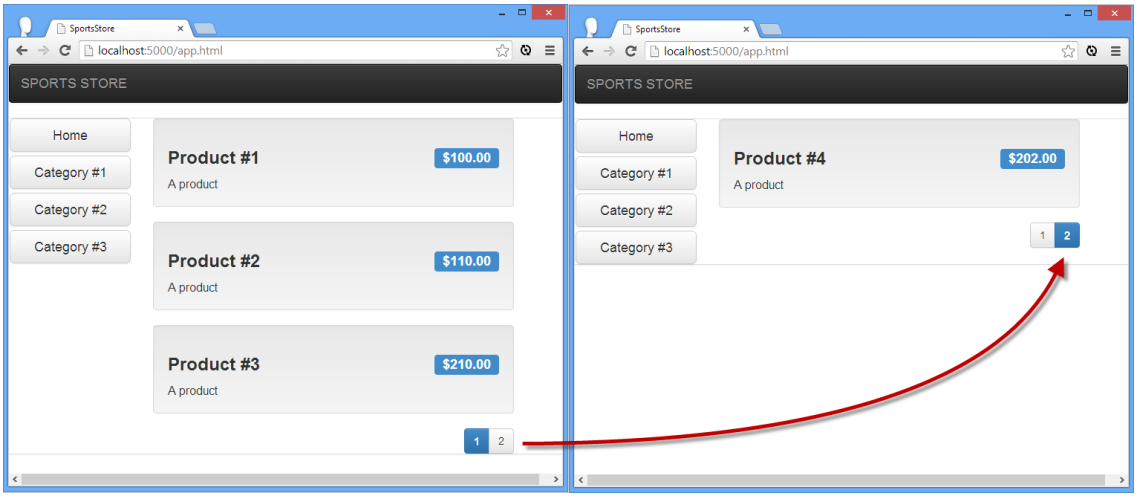

Figure 6-11. Paginating the product details

Summary

In this chapter, I started the process of developing the SportsStore application. All development frameworks that follow the MVC pattern have a common characteristic, which is that there is a lot of seemingly slow preparation and then, all of a sudden, features start to fall into place. AngularJS is no exception, and you can get a sense of the quickening pace throughout this chapter, to the point where adding pagination took longer for me to explain than to actually do. Now that the basic plumbing is in place, the pace will continue to crack along in the next chapter, where I will start using the real data from Parse.com, implement the shopping cart, and start the checkout process.

CHAPTER 7

SportsStore: Navigation and Checkout

In this chapter, I will continue the development of the SportsStore application by adding support for working with the real data, by implementing the cart, and by beginning work on the order checkout process.

Preparing the Example Project

I am going to continue building the project I started in Chapter 6. You can download the source code from Chapter 6 from `www.apress.com` if you want to follow along with the examples but don't want to have to build the project from scratch.

Using the Real Product Data

In Chapter 6, I put all of the features in place for displaying the product data to the user, but I did so using dummy data so that I could focus on building the basic plumbing of the application. It is now time to switch over to using the real data, which I will obtain from Parse.com and which I set up right at the start of Chapter 6.

AngularJS provides support for making Ajax requests through a service called `$http`. I describe how services work in detail in Part 3 and the `$http` service itself in Chapter 23, but you can get a sense of how it works through the changes I made to the top-level `sportsStoreCtrl` controller, as shown in Listing 7-1.

Listing 7-1. Making an Ajax Request in the sportsStore.js File

```
angular.module("sportsStore")
    .constant("dataUrl", "https://api.parse.com/1/classes/Products")
    .run(function ($http) {
        $http.defaults.headers.common["X-Parse-Application-Id"]
            = "Mc0i2Q6FZNovOr4bQD6QRFwzgw4iRHOy2JcDYuLq";
        $http.defaults.headers.common["X-Parse-REST-API-Key"]
            = "h4dnGScEtCWMMjaWmCJYNBZrDCrnROZmsKvEwXLD";
    })
    .controller("sportsStoreCtrl", function ($scope, $http, dataUrl) {
```

```
    $scope.data = {};

    $http.get(dataUrl)
        .success(function (data) {
            $scope.data.products = data.results;

        })
        .error(function (response) {
            $scope.data.error = response.error || response;
        });
});
```

Most JavaScript methods calls, including those made on AngularJS components, are *synchronous*, which means that execution doesn't move on to the next statement until the current one has been completed. That doesn't work when making network requests in web applications because we want the user to be able to interact with the application while the request is being made in the background.

I am going to obtain the data I need using an Ajax request. Ajax stands for *Asynchronous JavaScript and XML*, where the important word is *asynchronous*. An Ajax request is a regular HTTP request that happens asynchronously, in other words, in the background. AngularJS represents asynchronous operations using *promises*, which will be familiar to you if you have used libraries such as jQuery (and which I introduced in Chapter 5 and explain in detail in Chapter 20).

The `$http` service defines methods for making different kinds of Ajax request. The `get` method, which is the one I have used here, uses the HTTP GET method to request the URL passed as an argument. I have defined the URL as a constant called `dataUrl` and used the URL from Chapter 6 with which I tested `Parse.com`. I am able to omit the keys from the URL itself now that I am working with AngularJS and I have added them as default headers for Ajax requests using the `run` method (which I describe in Chapter 9).

The `$http.get` method starts the Ajax request, and execution of the application continues, even though the request has yet to be completed. AngularJS needs a way to notify me when the server has responded to the request, which is where the promise comes in. The `$http.get` method returns an object that defines `success` and `error` methods. I pass functions to these methods, and AngularJS *promises* to call one of them to tell me how the request turns out.

AngularJS will invoke the function I passed to the `success` method if everything with the HTTP request went well and—as a bonus—will automatically convert JSON data to JavaScript objects and pass them as the argument to the `success` function. If there is a problem with the Ajax HTTP request, then AngularJS will invoke the function I passed to the `error` method.

Tip JSON stands for *JavaScript Object Notation* and is a data exchange format that is widely used in web applications. JSON represents data in a way that is similar to JavaScript, which makes it easy to operate on JSON data in JavaScript applications. JSON has largely displaced XML, the *X* in Ajax, because it is human-readable and easy to implement. I introduced JSON in Chapter 5, and you can learn about the details of it at `http://en.wikipedia.org/wiki/Json`.

The `success` function I have used in the listing is simple because it relies on the automatic conversion that AngularJS performs for JSON data. I just assign the data that is obtained from the server to the `data.products` variable on the controller scope. The `error` function assigns the object passed by AngularJS to describe the problem to the `data.error` variable on the scope. (I'll return to the error in the next section.)

Tip Successful responses from `Parse.com` are an object with a `results` property that is set to the data that has been requested. Errors are expressed an object which has an `error` property. You can see how I have used the `results` and `error` properties in the listing. Not all errors will come from `Parse.com`, which is why I test for the `error` property and fallback to using the entire argument object.

You can see the effect of making the Ajax request in Figure 7-1. When AngularJS creates its instance of the `sportsStore` controller, the HTTP request is started, and then the scope is updated with the data when it arrives. The product detail, category, and page features that I created in Chapter 6 operate just as they did before, but with the product data delivered from Parse.com.

UNDERSTANDING THE SCOPE

It may not be obvious when testing the changes, but obtaining the data via Ajax highlights one of the most important aspects of AngularJS development, which is the dynamic nature of scopes. When the application first starts, the HTML content is generated and displayed to the user even though there is no product information available.

At some point after the content has been rendered, the data will arrive from the server and be assigned to the `data.products` variable in the scope. When this happens, AngularJS updates all of the bindings and the output from behaviors that depend on the product data, ensuring that the new data is propagated throughout the application. In essence, AngularJS scopes are *live* data stores, which respond and propagate changes. You will see countless examples of this propagation of changes throughout the book.

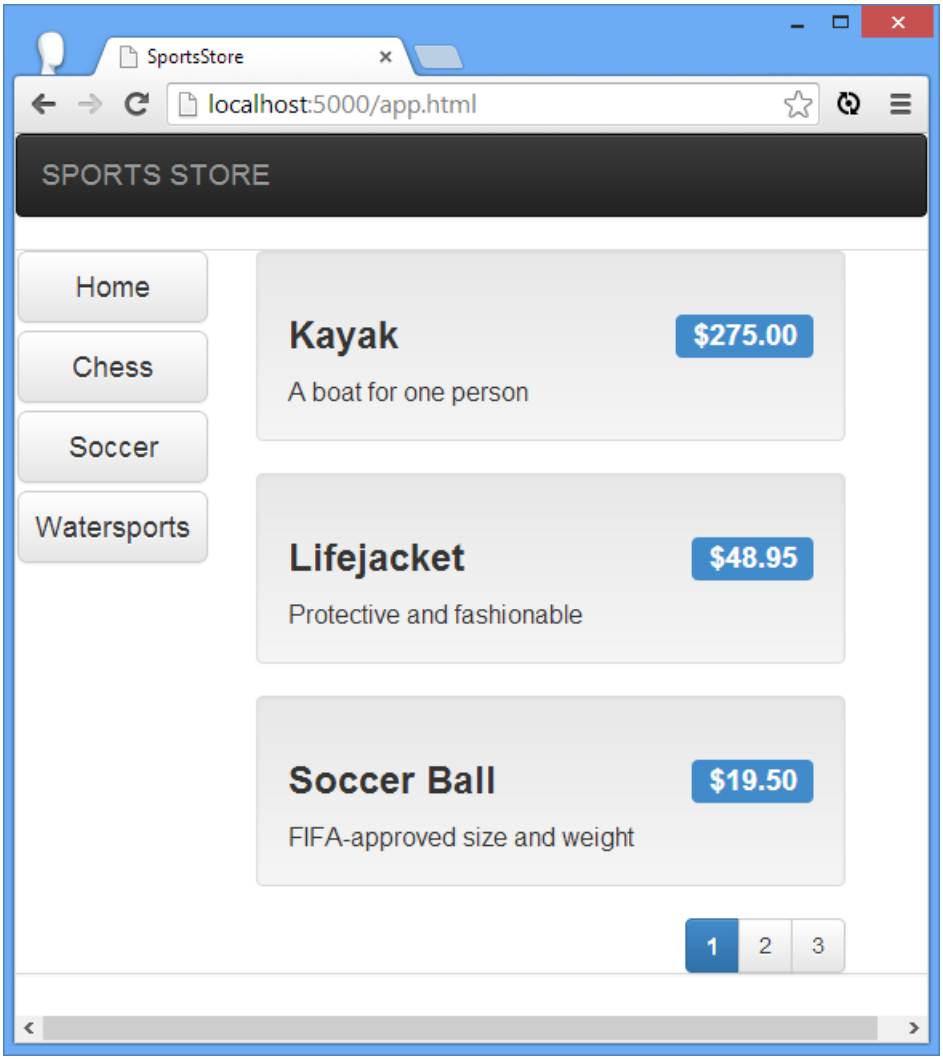

Figure 7-1. Obtaining product data via Ajax

Handling Ajax Errors

Dealing with successful Ajax requests is easy because I just assign the data to the scope and let AngularJS update all of the bindings and directives in the views. I have to work a little harder to deal with errors and add some new elements to the view that I will display when there is a problem. In Listing 7-2, you can see the changes that I have made to the `app.html` file to display errors to the user.

Listing 7-2. Displaying Errors in the app.html File

```
<!DOCTYPE html>
<html ng-app="sportsStore">
```

```
<head>
    <title>SportsStore</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStore", ["customFilters"]);
    </script>
    <script src="controllers/sportsStore.js"></script>
    <script src="filters/customFilters.js"></script>
    <script src="controllers/productListControllers.js"></script>
</head>
<body ng-controller="sportsStoreCtrl">
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>

    <div class="alert alert-danger" ng-show="data.error">
        Error ({{data.error.status}}). The product data was not loaded.
        <a href="/app.html" class="alert-link">Click here to try again</a>
    </div>

    <div class="panel panel-default row" ng-controller="productListCtrl"
        ng-hide="data.error">
        <div class="col-xs-3">
            <a ng-click="selectCategory()"
               class="btn btn-block btn-default btn-lg">Home</a>
            <a ng-repeat="item in data.products | orderBy:'category' | unique:'category'"
               ng-click="selectCategory(item)" class=" btn btn-block btn-default btn-lg"
               ng-class="getCategoryClass(item)">
                {{item}}
            </a>
        </div>
        <div class="col-xs-8">
            <div class="well"
                 ng-repeat=
             "item in data.products | filter:categoryFilterFn | range:selectedPage:pageSize">
                <h3>
                    <strong>{{item.name}}</strong>
                    <span class="pull-right label label-primary">
                        {{item.price | currency}}
                    </span>
                </h3>
                <span class="lead">{{item.description}}</span>
            </div>
            <div class="pull-right btn-group">
                <a ng-repeat=
                  "page in data.products | filter:categoryFilterFn | pageCount:pageSize"
                   ng-click="selectPage($index + 1)" class="btn btn-default"
                   ng-class="getPageClass($index + 1)">
                    {{$index + 1}}
                </a>
            </div>
        </div>
    </div>
</body>
</html>
```

I have added a new `div` element to the view, which shows an error to the user. I have used the `ng-show` directive, which hides the element it applied to until the expression specified in the attribute value evaluates to `true`. I have specified the `data.error` property, which AngularJS takes as an instruction to show the `div` element when the property has been assigned a value. Since the `data.error` property is `undefined` until an Ajax error occurs, the visibility of the `div` element is tied to the outcome of the `$http.get` method in the controller.

The counterpart to the `ng-show` directive is `ng-hide`, which I have applied to the `div` element that contains the category buttons and the product details. The `ng-hide` directive will show an element and its contents until its expression evaluates to `true`, at which point they will be hidden. The overall effect is that when there is an Ajax error, the normal content is hidden and replaced with the error, as shown in Figure 7-2.

Tip I describe the `ng-show` and `ng-hide` directives in detail in Chapter 10.

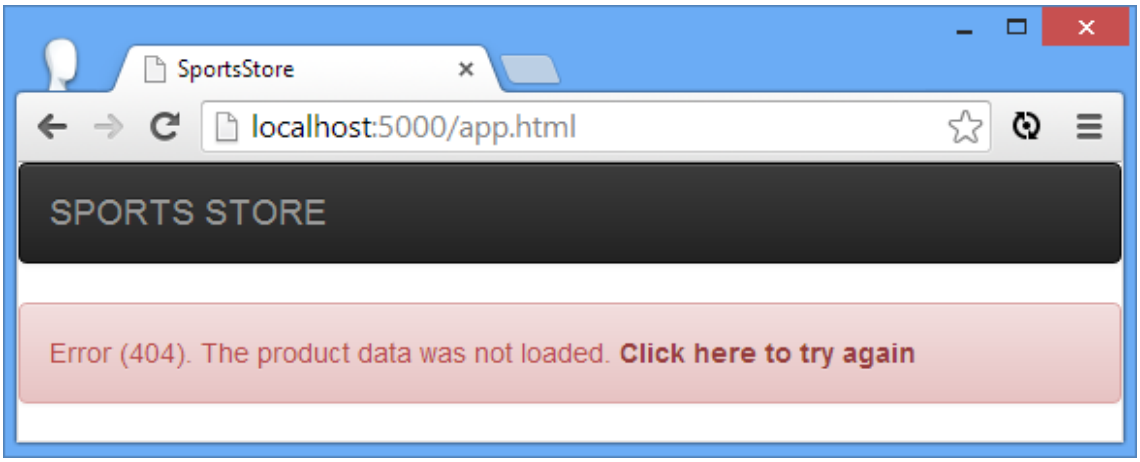

Figure 7-2. Displaying an error to the user

Tip I created this screenshot by changing the value of the `dataUrl` in the `sportsStore.js` file to one that doesn't exist.

The object passed to the `error` function defines `status` and `message` properties. The status property is set to the HTTP error code, and the message property returns a string that describes the problem. I included the `status` property in the message that I show to the user, along with a link that lets them reload the application and, implicitly, try to load the data again.

Creating Partial Views

The HTML in the `app.html` file is approaching the point of complexity where it isn't immediately obvious what every element does—something that will get worse as I add further features to the SportsStore application.

Fortunately, I can break up the markup into separate files and use the `ng-include` directive to import those files at runtime. To that end, I created the `views/productList.html` file, the contents of which are shown in Listing 7-3.

Listing 7-3. The Contents of the productList.html File

```
<div class="panel panel-default row" ng-controller="productListCtrl"
        ng-hide="data.error">
    <div class="col-xs-3">
        <a ng-click="selectCategory()"
           class="btn btn-block btn-default btn-lg">Home</a>
        <a ng-repeat="item in data.products | orderBy:'category' | unique:'category'"
           ng-click="selectCategory(item)" class=" btn btn-block btn-default btn-lg"
           ng-class="getCategoryClass(item)">
            {{item}}
        </a>
    </div>
    <div class="col-xs-8">
        <div class="well"
             ng-repeat=
         "item in data.products | filter:categoryFilterFn | range:selectedPage:pageSize">
            <h3>
                <strong>{{item.name}}</strong>
                <span class="pull-right label label-primary">
                    {{item.price | currency}}
                </span>
            </h3>
            <span class="lead">{{item.description}}</span>
        </div>
        <div class="pull-right btn-group">
            <a ng-repeat=
               "page in data.products | filter:categoryFilterFn | pageCount:pageSize"
               ng-click="selectPage($index + 1)" class="btn btn-default"
               ng-class="getPageClass($index + 1)">
                {{$index + 1}}
            </a>
        </div>
    </div>
</div>
```

I have copied the elements that define the product and category lists into the HTML file. Partial views are fragments of HTML, which means that they do not require `html`, `head`, and `body` elements in the way that a complete HTML document does. In Listing 7-4, you can see how I have removed these elements from the `app.html` file and replaced them with the `ng-include` directive.

Tip There are three benefits to using partial views. The first is to break up the application into manageable chunks, as I have done here. The second is to create fragments of HTML that can be used repeatedly in an application. The third is to make it easier to show different areas of functionality to the user as they use the application—I'll return to this benefit in the "Defining URL Routes" section later in the chapter.

Listing 7-4. Importing a Partial View in the app.html File

```
<!DOCTYPE html>
<html ng-app="sportsStore">
<head>
    <title>SportsStore</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStore", ["customFilters"]);
    </script>
    <script src="controllers/sportsStore.js"></script>
    <script src="filters/customFilters.js"></script>
    <script src="controllers/productListControllers.js"></script>
</head>
<body ng-controller="sportsStoreCtrl">
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>

    <div class="alert alert-danger" ng-show="data.error">
        Error ({{data.error.status}}). The product data was not loaded.
        <a href="/app.html" class="alert-link">Click here to try again</a>
    </div>

    <ng-include src="'views/productList.html'"></ng-include>

</body>
</html>
```

The creator of a directive can specify how it can be applied: as an element, as an attribute, as a class, or even as an HTML comment. I explain how this is done in Chapter 16, but the `ng-include` directive has been set up so that it can be applied as an element and as the more conventional attribute, and I have used it in this way solely for variety. When AngularJS encounters the `ng-include` directive, it makes an Ajax request, loads the file specified by the `src` attribute, and inserts the contents in place of the element. There is no visible difference in the content presented to the user, but I have simplified the markup in the `app.html` file and put all the product list–related HTML in a separate file.

Tip When using the `ng-include` directive, I specified the name of the file as a literal value in single quotes. If I had not done this, then the directive would have looked for a scope property to get the name of the file.

Creating the Cart

The user can see the products that I have available, but I can't sell anything without a shopping cart. In this section, I will build the cart functionality that will be familiar to anyone who has used an e-commerce site, the basic flow of which is illustrated by Figure 7-3.

Figure 7-3. The basic flow of the shopping cart

As you will see in the following sections, several sets of changes are required to implement the cart feature, including creating a custom AngularJS component.

Defining the Cart Module and Service

So far, I have been organizing the files in my project based on the type of component they contain: Filters are defined in the `filters` folder, views in the `views` folder, and so on. This makes sense when building the basic features of an application, but there will always be some functionality in a project that is relatively self-contained but requires a mix of AngularJS components. You can continue to organize the files by component type, but I find it more useful to order the files by the function that they collectively represent, for which I use the `components` folder. The cart functionality is suitable for this kind of organization because, as you will see, I am going to need partial views and several components to get the effect I require. I started by creating the `components/cart` folder and adding a new JavaScript file to it called `cart.js`. You can see the contents of this file in Listing 7-5.

Listing 7-5. The Contents of the cart.js File

```
angular.module("cart", [])
.factory("cart", function () {

    var cartData = [];

    return {

        addProduct: function (id, name, price) {
            var addedToExistingItem = false;
            for (var i = 0; i < cartData.length; i++) {
                if (cartData[i].objectId == id) {
                    cartData[i].count++;
                    addedToExistingItem = true;
                    break;
```

```
                }
            }
            if (!addedToExistingItem) {
                cartData.push({
                    count: 1, objectId: id, price: price, name: name
                });
            }
        },

        removeProduct: function (id) {
            for (var i = 0; i < cartData.length; i++) {
                if (cartData[i].objectId == id) {
                    cartData.splice(i, 1);
                    break;
                }
            }
        },

        getProducts: function () {
            return cartData;
        }
    }
});
```

I started by creating a custom service in a new module called `cart`. AngularJS provides a lot of its functionality through services, but they are simply singleton objects that are accessible throughout an application. (*Singleton* just means that only one object will be created and shared by all of the components that depend on the service.)

Not only does using a service allow me to demonstrate an important AngularJS feature, but implementing the cart this way works well because having a shared instance ensures that every component can access the cart and have the same view of the user's product selections.

As I explain in Chapter 18, there are different ways to create services depending on what you are trying to achieve. I have used the simplest in Listing 7-5, which is to call the `Module.factory` method and pass in the name of the service (which is `cart`, in this case) and a factory function. The factory function will be invoked when AngularJS needs the service and is responsible for creating the service object; since one service object is used throughout the application, the factory function will be called only once.

My `cart` service factory function returns an object with three methods that operate on a data array that is not exposed directly through the service, which I did to demonstrate that you don't have to expose all of the workings in a service. The `cart` service object defines the three methods described in Table 7-1. I represent products in the cart with objects that define `id`, `name`, and `price` properties to describe the product and a `count` property to record the number the user has added to the basket.

Table 7-1. *The Methods Defined by the Cart Service*

Method	Description
`addProduct(id, name, price)`	Adds the specified product to the cart or increments the number required if the cart already contains the product

`removeProduct(id)`	Removes the product with the specified ID
`getProducts()`	Returns the array of objects in the cart

Creating a Cart Widget

My next step is to create a widget that will summarize the contents of the cart and provide the user with the means to begin the checkout process, which I am going to do by creating a custom directive. *Directives* are self-contained, reusable units of functionality that sit at the heart of AngularJS development. As you start with AngularJS, you will rely on the many built-in directives (which I describe in Chapters 9–12), but as you gain confidence, you will find yourself creating custom directives to tailor functionality to suit your applications.

You can do a lot with directives, which is why it takes me six chapters to describe them fully later in the book. They even support a cut-down version of jQuery, called *jqLite*, to manipulate elements in the DOM. In short, directives allow you to write anything from simple helpers to complex features and to decide whether the result is tightly woven into the current application or completely reusable in other applications. Listing 7-6 shows the additions I made to the `cart.js` file to create the widget directive, which is at the simpler end of what you can do with directives.

Listing 7-6. Adding a Directive to the cart.js File

```
angular.module("cart", [])
.factory("cart", function () {

    var cartData = [];

    return {
        // ...service statements omitted for brevity...
    }
})
.directive("cartSummary", function (cart) {
    return {
        restrict: "E",
        templateUrl: "components/cart/cartSummary.html",
        controller: function ($scope) {

            var cartData = cart.getProducts();

            $scope.total = function () {
                var total = 0;
                for (var i = 0; i < cartData.length; i++) {
                    total += (cartData[i].price * cartData[i].count);
                }
                return total;
            }

            $scope.itemCount = function () {
                var total = 0;
                for (var i = 0; i < cartData.length; i++) {
```

```
                total += cartData[i].count;
            }
            return total;
        }
    }
};
});
```

Directives are created by calling the `directive` method on an AngularJS module and passing in the name of the directive (`cartSummary` in this case) and a factory function that returns a *directive definition object*. The definition object defines properties that tell AngularJS what your directive does and how it does it. I have specified three properties when defining the `cartSummary` directive, and I have described them briefly in Table 7-2. (I describe and demonstrate the complete set of properties in Chapters 16 and 17.)

Tip Although my directive is rather basic, it isn't the simplest approach you can use to create a directive. In Chapter 15, I show you how to create directives that use jqLite, the AngularJS version of jQuery to manipulate existing content. The kind of directive that I have created here, which specifies a template and a controller and restricts how it can be applied, is covered in Chapter 16 and Chapter 17.

Table 7-2. *The Definition Properties Used for the cartSummary Directive*

Name	Description
`restrict`	Specifies how the directive can be applied. I have used a value of `E`, which means that this directive can be applied only as an element. The most common value is `EA`, which means that the directive can be applied as an element or as an attribute.
`templateUrl`	Specifies the URL of a partial view whose contents will be inserted into the directive's element.
`controller`	Specifies a controller that will provide data and behaviors to the partial view.

In short, my directive definition defines a controller, tells AngularJS to use the `components/cart/cartSummary.html` view, and restricts the directive so that it can be applied only as an element. Notice that the controller in Listing 7-6 declares a dependency on the `cart` service, which is defined in the same module. This allows me to define the `total` and `itemCount` behaviors that consume the methods provided by the service to operate on the cart contents. The behaviors defined by the controller are available to the partial view, which is shown in Listing 7-7.

Tip This partial view contains a `style` element to redefine some of the Bootstrap CSS for the navigation bar that runs across the top of the SportsStore layout. I don't usually like embedding `style` elements in partial views,

but I do so when the changes affect only that view and there is a small amount of CSS. In all other situations, I would define a separate CSS file and import it into the application's main HTML file.

Listing 7-7. The Contents of the cartSummary.html File

```
<style>
    .navbar-right { float: right !important; margin-right: 5px;}
    .navbar-text { margin-right: 10px; }
</style>

<div class="navbar-right">
    <div class="navbar-text">
        <b>Your cart:</b>
        {{itemCount()}} item(s),
        {{total() | currency}}
    </div>
    <a class="btn btn-default navbar-btn">Checkout</a>
</div>
```

The partial view uses the controller behaviors to display the number of items and the total value of those items. There is also an `a` element that is labeled Checkout; clicking the button doesn't do anything at the moment, but I'll wire it up later in the chapter.

Applying the Cart Widget

Applying the cart widget to the application requires three steps: adding a `script` element to import the contents of the JavaScript file, adding a dependency for the `cart` module, and adding the directive element to the markup. Listing 7-8 shows all three changes applied to the `app.html` file.

Listing 7-8. Adding the Cart Widget to the app.html File

```
<!DOCTYPE html>
<html ng-app="sportsStore">
<head>
    <title>SportsStore</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStore", ["customFilters", "cart"]);
    </script>
    <script src="controllers/sportsStore.js"></script>
    <script src="filters/customFilters.js"></script>
    <script src="controllers/productListControllers.js"></script>
    <script src="components/cart/cart.js"></script>
</head>
<body ng-controller="sportsStoreCtrl">
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
        <cart-summary />
    </div>
```

```
    <div class="alert alert-danger" ng-show="data.error">
        Error ({{data.error.status}}). The product data was not loaded.
        <a href="/app.html" class="alert-link">Click here to try again</a>
    </div>
    <ng-include src="'views/productList.html'"></ng-include>
</body>
</html>
```

Notice that although I used the name `cartSummary` when I defined the directive in Listing 7-8, the element I added to the `app.html` file is `cart-summary`. AngularJS *normalizes* component names to map between these formats, as I explain in Chapter 15. You can see the effect of the cart summary widget in Figure 7-4. The widget doesn't do much at the moment, but I'll start adding other features that will drive its behavior in the following sections.

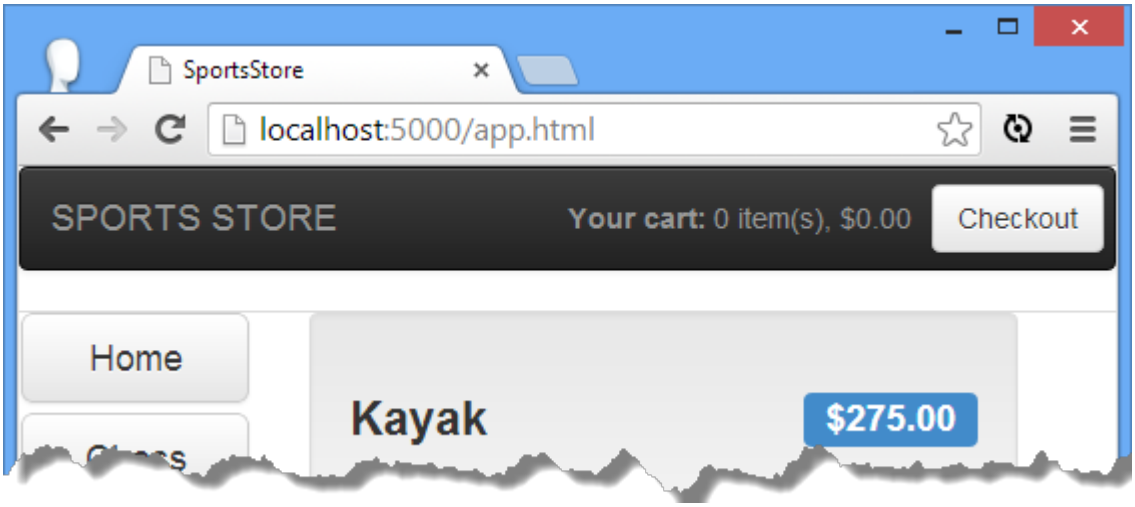

Figure 7-4. The cart summary widget

Adding Product Selection Buttons

As with all AngularJS development, there is some up-front effort to develop the foundations and then other features start to snap into place—something that holds true for the cart as much as another part of the application. My next step is to add buttons to the product details so that the user can add products to the cart. First, I need to add a behavior to the controller for the product list view to operate on the cart. Listing 7-9 shows the changes I have made to the `controllers/productListController.js` file.

Listing 7-9. Adding Support for the Cart to the productListControllers.js File

```
angular.module("sportsStore")
    .constant("productListActiveClass", "btn-primary")
    .constant("productListPageCount", 3)
    .controller("productListCtrl", function ($scope, $filter,
        productListActiveClass, productListPageCount, cart) {

        var selectedCategory = null;
```

```
        $scope.selectedPage = 1;
        $scope.pageSize = productListPageCount;

        $scope.selectCategory = function (newCategory) {
            selectedCategory = newCategory;
            $scope.selectedPage = 1;
        }

        $scope.selectPage = function (newPage) {
            $scope.selectedPage = newPage;
        }

        $scope.categoryFilterFn = function (product) {
            return selectedCategory == null ||
                product.category == selectedCategory;
        }

        $scope.getCategoryClass = function (category) {
            return selectedCategory == category ? productListActiveClass : "";
        }

        $scope.getPageClass = function (page) {
            return $scope.selectedPage == page ? productListActiveClass : "";
        }

        $scope.addProductToCart = function (product) {
            cart.addProduct(product.objectId, product.name, product.price);
        }
    });
```

I have declared a dependency on the `cart` service and defined a behavior called `addProductToCart` that takes a product object and uses it to call the `addProduct` method on the `cart` service.

Tip This pattern of declaring a dependency on a service and then selectively exposing its functionality through the scope is one you will encounter a lot in AngularJS development. Views can access only the data and behaviors that are available through the scope—although, as I demonstrated in Chapter 6 (and explain in depth in Chapter 13), scopes can inherit from one another when controllers are nested or (as I explain in Chapter 17) when directives are defined.

I can then add `button` elements to the partial view that displays the product details and invokes the `addProductToCart` behavior, as shown in Listing 7-10.

Tip Bootstrap lets me style `a` and `button` elements so they have the same appearance; as a consequence, I tend to use them interchangeably. That said, `a` elements are more useful when using *URL routing*, which I describe later in this chapter.

Listing 7-10. Adding Buttons to the productList.html File

```
<div class="panel panel-default row" ng-controller="productListCtrl"
        ng-hide="data.error">
    <div class="col-xs-3">
        <a ng-click="selectCategory()"
           class="btn btn-block btn-default btn-lg">Home</a>
        <a ng-repeat="item in data.products | orderBy:'category' | unique:'category'"
           ng-click="selectCategory(item)" class=" btn btn-block btn-default btn-lg"
           ng-class="getCategoryClass(item)">
            {{item}}
        </a>
    </div>
    <div class="col-xs-8">
        <div class="well"
             ng-repeat=
         "item in data.products | filter:categoryFilterFn | range:selectedPage:pageSize">
            <h3>
                <strong>{{item.name}}</strong>
                <span class="pull-right label label-primary">
                    {{item.price | currency}}
                </span>
            </h3>
            <button ng-click="addProductToCart(item)"
                    class="btn btn-success pull-right">
                Add to cart
            </button>
            <span class="lead">{{item.description}}</span>
        </div>
        <div class="pull-right btn-group">
            <a ng-repeat=
                  "page in data.products | filter:categoryFilterFn | pageCount:pageSize"
               ng-click="selectPage($index + 1)" class="btn btn-default"
               ng-class="getPageClass($index + 1)">
                {{$index + 1}}
            </a>
        </div>
    </div>
</div>
```

You can see the buttons and the effect they have in Figure 7-5. Clicking one of the Add to cart buttons invokes the controller behavior, which invokes the service methods, which then causes the cart summary widget to update.

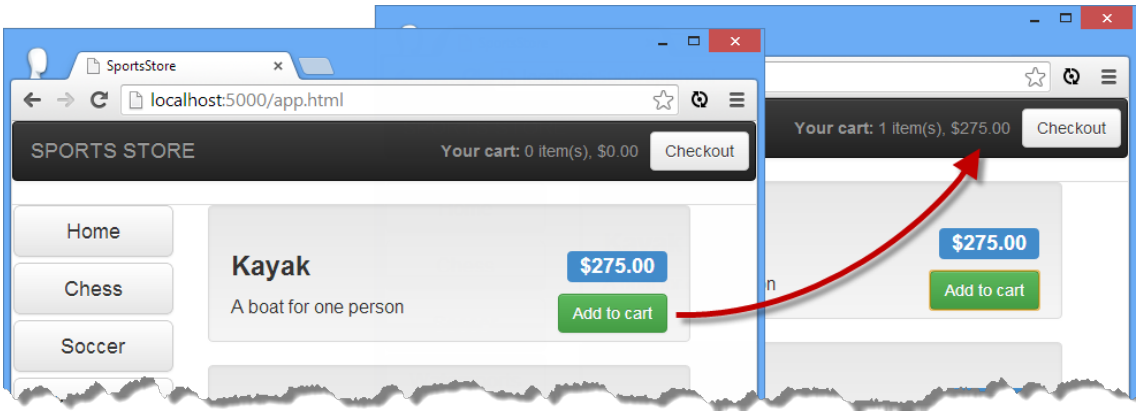

Figure 7-5. Adding products to the cart

Adding URL Navigation

Before I go any further and add support for checking out, I am going to enhance the infrastructure of the SportsStore application by adding support for *URL routing*. I describe URL routing in detail in Chapter 22, but the short version is that it allows for different partial views to be displayed automatically based on the current URL. This makes it easier to build larger applications that the user can navigate freely around, and I will use it as the foundation for displaying the views that the user needs to complete their purchase and submit an order to the server.

To get started, I need to create a view that I will display when the user begins the checkout process. Listing 7-11 shows the contents of the `views/checkoutSummary.html` file, which contains some placeholder content for the moment. I'll return to this file and add the real content once I have set up the URL routing feature.

Listing 7-11. The Contents of the checkoutSummary.html File

```
<div class="lead">
    This is the checkout summary view
</div>
<a href="#/products" class="btn btn-primary">Back</a>
```

Defining URL Routes

I am going to start by defining the *routes* I require, which are the mappings between specific URLs and the views that should be displayed when the browser navigates to that URL. The first two will map the `/product` and `/checkout` URLs to the `productList.html` and `checkoutSummary.html` views, respectively. The other will be a catchall route that will display the `productList.html` view by default. Listing 7-12 shows the changes I have made to implement routing in the `app.html` file.

Listing 7-12. Adding Support for URL Routing in the app.html File

```
<!DOCTYPE html>
<html ng-app="sportsStore">
<head>
    <title>SportsStore</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStore", ["customFilters", "cart", "ngRoute"])
        .config(function ($routeProvider) {

            $routeProvider.when("/checkout", {
                templateUrl: "/views/checkoutSummary.html"
            });

            $routeProvider.when("/products", {
                templateUrl: "/views/productList.html"
            });

            $routeProvider.otherwise({
                templateUrl: "/views/productList.html"
            });
        });
    </script>
    <script src="controllers/sportsStore.js"></script>
    <script src="filters/customFilters.js"></script>
    <script src="controllers/productListControllers.js"></script>
    <script src="components/cart/cart.js"></script>
    <script src="ngmodules/angular-route.js"></script>
</head>
<body ng-controller="sportsStoreCtrl">
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
        <cart-summary />
    </div>
    <div class="alert alert-danger" ng-show="data.error">
        Error ({{data.error.status}}). The product data was not loaded.
        <a href="/app.html" class="alert-link">Click here to try again</a>
    </div>
    <ng-view />
</body>
</html>
```

I have added a `script` element to import the `angular-route.js` file into the application. The functionality that this file provides is defined in a module called `ngRoute`, which I have declared as a dependency of the `sportsStore` module.

To set up my routes, I have called the `config` method on the module object. The `config` method takes a function as its argument, which is executed when the module is loaded but before the application is executed, providing an opportunity for any one-off configuration tasks.

The function that I passed to the `config` method declares a dependency on a *provider*. As I mentioned earlier, there are different ways to create AngularJS services, and one of them creates a service that can be configured through a *provider object*, whose name is the concatenation of the service name and `Provider`. The `$routeProvider` that I have declared a dependency on is the provider for the `$route` service and is used to set up the URL routing in an application.

Tip I explain how to create services with providers in Chapter 18 and how to use the `$route` service and the `$routeProvider` in Chapter 22.

I use two methods defined by the `$routeProvider` object to set up the routes I require. The `when` method allows me to match a URL to a view, like this:

```
...
$routeProvider.when("/checkout", {
    templateUrl: "/views/checkoutSummary.html"
});
...
```

This statement tells AngularJS that when the URL is `/checkout`, I want the `/views/checkoutSummary.html` file to be displayed. The `otherwise` method specifies the view that should be used when the URL doesn't match one of those defined by the `when` method. It is always sensible to define such a fallback route, and mine specifies the `/views/ProductList.html` view file.

URL routes are matched against the *path* section of the current URL and *not* the complete URL. Here is a URL that would match the route shown earlier:

```
http://localhost:5000/app.html#/checkout
```

I have highlighted the path, which follows the `#` character in the URL. AngularJS doesn't monitor the whole URL because a URL such as `http://localhost:5000/checkout` would cause the browser to dump the AngularJS application and try to load a different document from the server—something that is rarely required. This point causes a lot of confusion, so I have summarized the effect of my URL routing policy in Table 7-3.

Tip As I describe in Chapter 22, you can enable support for using the HTML5 History API, which changes the way URLs are monitored so that something like `http://localhost:5000/checkout` will work. Caution is required because browser implementations differ, and it is easy to confuse the user because the browser will attempt to load a different document if they try to edit the URL manually.

Table 7-3. *The Effect of the URL Routing Policy*

URL	Effect
`http://localhost:5000/app.html#/checkout`	Displays the `checkoutSummary.html` view
`http://localhost:5000/app.html#/products`	Displays the `productList.html` view
`http://localhost:5000/app.html#/other`	Displays the `productList.html` view (because of the

	fallback route defined by the `otherwise` method)
`http://localhost:5000/app.html`	Displays the `productList.html` view (because of the fallback route defined by the `otherwise` method)

Displaying the Routed View

The routing policy defines which views should be displayed for given URL paths, but it doesn't tell AngularJS *where* to display them. For that I need the `ng-view` directive, which is defined in the `ngRoute` module along with the other routing features. In Listing 7-12, I replaced the `ng-include` directive with `ng-view`, as follows:

```
...
<body ng-controller="sportsStoreCtrl">
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
        <cart-summary />
    </div>
    <div class="alert alert-danger" ng-show="data.error">
        Error ({{data.error.status}}). The product data was not loaded.
        <a href="/app.html" class="alert-link">Click here to try again</a>
    </div>
    <ng-view />
</body>
...
```

There are no configuration options or settings required; just adding the directive tells AngularJS where it should insert the content of the currently selected view.

Using URL Routing to Navigate

Having defined my URL routes and applied the `ng-view` directive, I can change the URL path to navigate through the application. My first change is to the Checkout button displayed by the cart summary widget that I created earlier in the chapter. Listing 7-13 shows the change I made to the `cartSummary.html` file.

Listing 7-13. Using URL Path Navigation to the cartSummary.html File

```
<style>
    .navbar-right { float: right !important; margin-right: 5px;}
    .navbar-text { margin-right: 10px; }
</style>

<div class="navbar-right">
    <div class="navbar-text">
        <b>Your cart:</b>
        {{itemCount()}} item(s),
        {{total() | currency}}
    </div>
    <a href="#/checkout" class="btn btn-default navbar-btn">Checkout</a>
</div>
```

I updated the a element to add an href attribute whose value changes the path. Clicking the element will cause the browser to navigate to the new URL (which is local to the already-loaded document). The navigation change is detected by the AngularJS routing service, which causes the ng-view directive to display the checkoutSummary.html view, as illustrated by Figure 7-6.

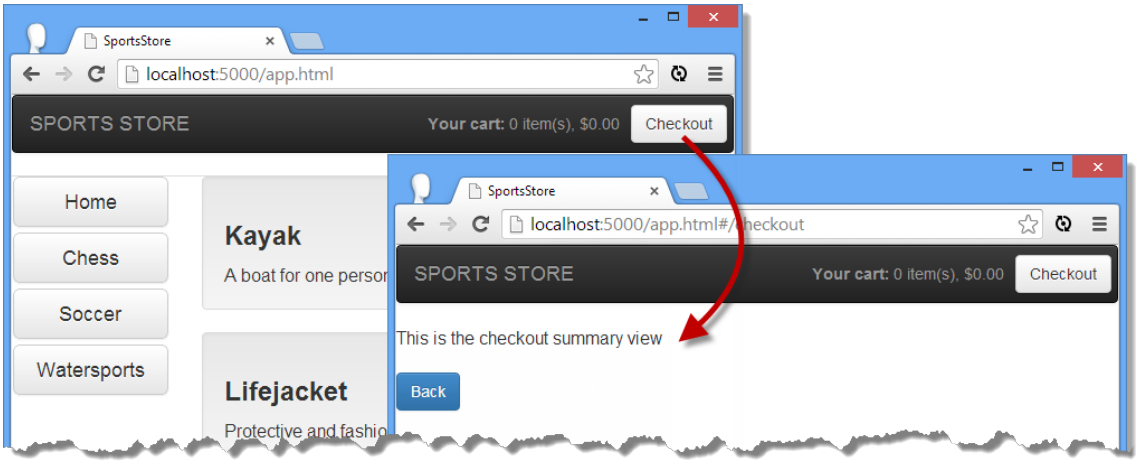

Figure 7-6. Navigating to the checkout summary

Notice that the URL displayed by the browser changes from the initial starting point of http://localhost:5000/app.html to http://localhost:5000/app.html#/checkout. You can click the Back button displayed by the checkoutSummary.html view, which I configured in Listing 7-11 to move to the /products path, as follows:

```
...
<a href="#/products" class="btn btn-primary">Back</a>
...
```

The main benefit of using URL routing is that components can change the layout shown by the ng-view directive without having any prior knowledge of the view that will be shown, the location or disposition of the ng-view directive, or sharing components (such as controllers or services) with the view that will be displayed. This makes it easier to scale up complex applications and makes it possible to change the behavior of the application just by changing the URL routing configuration.

Tip You can also return to the products listing by manually editing the URL to be http://localhost:5000/app.html#/products or http://localhost:5000/app.html#. Note the trailing # character in that last URL. If you omit it, the browser will interpret the URL as a request to load the app.html page, which will cause any unsaved state to be lost. For the SportsStore application, that means the contents of the cart will be lost. The finicky nature of the URLs means that the user can edit them directly but that the results can be unexpected with even the slightest error.

Starting the Checkout Process

Now that the routing configuration is in place, I am going to turn to the checkout process. My first task is to define a new controller called `cartSummaryController`, which I have placed in a new file called `controllers/checkoutControllers.js`. Listing 7-14 shows the contents of the new file.

Listing 7-14. The Contents of the checkoutControllers.js File

```
angular.module("sportsStore")
.controller("cartSummaryController", function($scope, cart) {

    $scope.cartData = cart.getProducts();

    $scope.total = function () {
        var total = 0;
        for (var i = 0; i < $scope.cartData.length; i++) {
            total += ($scope.cartData[i].price * $scope.cartData[i].count);
        }
        return total;
    }

    $scope.remove = function (id) {
        cart.removeProduct(id);
    }
});
```

The new controller is added to the `sportsStore` module and depends on the `cart` service. It exposes the contents of the cart through a scope property called `cartData` and defines behaviors to calculate the total value of the products in the cart and to remove a product from the cart. Using the features created by the controller, I can replace the temporary content in the `checkoutSummary.html` file with a summary of the cart. Listing 7-15 shows the changes I have made.

Listing 7-15. Revising the Contents of the checkoutSummary.html File

```
<h2>Your cart</h2>

<div ng-controller="cartSummaryController">

    <div class="alert alert-warning" ng-show="cartData.length == 0">
        There are no products in your shopping cart.
        <a href="#/products" class="alert-link">Click here to return to the catalogue</a>
    </div>

    <div ng-hide="cartData.length == 0">
        <table class="table">
            <thead>
                <tr>
                    <th>Quantity</th>
                    <th>Item</th>
                    <th class="text-right">Price</th>
                    <th class="text-right">Subtotal</th>
                </tr>
            </thead>
```

```
            <tbody>
                <tr ng-repeat="item in cartData">
                    <td class="text-center">{{item.count}}</td>
                    <td class="text-left">{{item.name}}</td>
                    <td class="text-right">{{item.price | currency}}</td>
                    <td class="text-right">{{ (item.price * item.count) | currency}}</td>
                    <td>
                        <button ng-click="remove(item.objectId)"
                                class="btn btn-sm btn-warning">Remove</button>
                    </td>
                </tr>
            </tbody>
            <tfoot>
                <tr>
                    <td colspan="3" class="text-right">Total:</td>
                    <td class="text-right">
                        {{total() | currency}}
                    </td>
                </tr>
            </tfoot>
        </table>

        <div class="text-center">
            <a class="btn btn-primary" href="#/products">Continue shopping</a>
            <a class="btn btn-primary" href="#/placeorder">Place order now</a>
        </div>
    </div>
</div>
```

There are no new techniques in this view. The controller is specified using the `ng-controller` directive, and I use the `ng-show` and `ng-hide` directives to show a warning when there are no items in the cart and a summary when there are. The `ng-repeat` directive is used to generate rows in a table for each product in the cart, and the details are displayed using data bindings. Each row contains unit and total pricing and a `button` that uses the `ng-click` directive to invoke the `remove` controller behavior and remove an item from the cart.

The two `a` elements at the end of the view allow the user to navigate elsewhere in the application:

```
...
<a class="btn btn-primary" href="#/products">Continue shopping</a>
<a class="btn btn-primary" href="#/placeorder">Place order now</a>
...
```

The Continue shopping button returns the user to the product list by navigating to the `#/products` path, and the Place order button navigates to a new URL path, `#/placeorder`, which I will configure in the next section.

Applying the Checkout Summary

The next step is to add a `script` element to the `app.html` file and define the additional routes that I will need to complete the checkout process, as shown in Listing 7-16.

Listing 7-16. Applying the Checkout Summary to the app.html File

```
<!DOCTYPE html>
<html ng-app="sportsStore">
<head>
    <title>SportsStore</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStore", ["customFilters", "cart", "ngRoute"])
        .config(function ($routeProvider) {

            $routeProvider.when("/complete", {
                templateUrl: "/views/thankYou.html"
            });

            $routeProvider.when("/placeorder", {
                templateUrl: "/views/placeOrder.html"
            });

            $routeProvider.when("/checkout", {
                templateUrl: "/views/checkoutSummary.html"
            });

            $routeProvider.when("/products", {
                templateUrl: "/views/productList.html"
            });

            $routeProvider.otherwise({
                templateUrl: "/views/productList.html"
            });
        });
    </script>
    <script src="controllers/sportsStore.js"></script>
    <script src="filters/customFilters.js"></script>
    <script src="controllers/productListControllers.js"></script>
    <script src="components/cart/cart.js"></script>
    <script src="ngmodules/angular-route.js"></script>
    <script src="controllers/checkoutControllers.js"></script>
</head>
<body ng-controller="sportsStoreCtrl">
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
        <cart-summary />
    </div>
    <div class="alert alert-danger" ng-show="data.error">
        Error ({{data.error.status}}). The product data was not loaded.
        <a href="/app.html" class="alert-link">Click here to try again</a>
    </div>
    <ng-view />
</body>
</html>
```

The new routes associate URLs with views that I will create in the next chapter. Figure 7-7 shows the cart summary that is now presented when the user clicks the Checkout button on the cart widget.

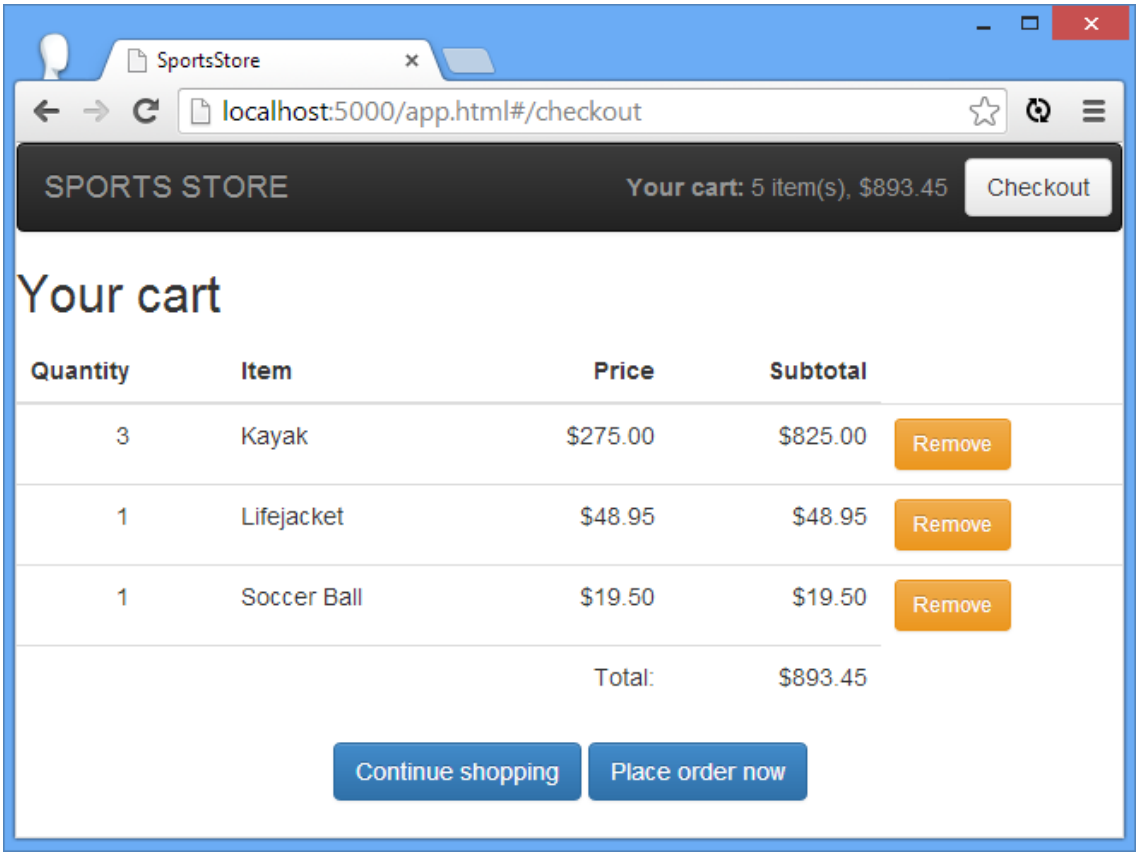

Figure 7-7. Summarizing the contents of the shopping cart

Summary

In this chapter, I continued the development of the SportsStore application to obtain the product data from Parse.com service, to add support for working with partial views, and to implement a custom directive. I also set up URL routing and started adding the functionality that will allow the user to place an order. In the next chapter, I will complete the SportsStore application and add support for administration.

CHAPTER 8

SportsStore: Orders and Administration

In this chapter, I complete the SportsStore application by collecting and validating shipping details and storing the order on Parse.com. I also build an administration application that allows authenticated users to see the set of orders and manage the product catalog.

Preparing the Example Project

I am going to continue to build on the project that I started in Chapter 6 and extended in Chapter 7. You can download the source code from Chapter 7 from `www.apress.com` if you want to follow along with the examples but don't want to have to build the project from scratch.

In Chapter 7, I started the checkout process by displaying a summary of the cart to the user. That summary included an `a` element that navigated to the `/placeorder` URL path, for which I added a URL route to the `app.html` file. In fact, I defined two routes, both of which I will need to complete the checkout process in this chapter:

```
...
$routeProvider.when("/complete", {
    templateUrl: "/views/thankYou.html"
});

$routeProvider.when("/placeorder", {
    templateUrl: "/views/placeOrder.html"
});
...
```

In this chapter I am going to create the views named in the URL routes and create the components required to complete the checkout process.

Getting Shipping Details

After showing the user a summary of the products in the cart, I want to capture the shipping details for the order. That takes me to the AngularJS features for working with forms, which you are likely to require in most web applications. I have created the `views/placeOrder.html` file to capture the user's shipping details, which is the view named in one of the routing URLs

shown earlier. I am going to introduce a number of form-related features, and to avoid having to repeat largely similar code, I am going to start working with a couple of data properties (for the user's name and street address) and then add other properties when I have introduced the features I will be using. Listing 8-1 shows the initial content of the `placeOrder.html` view file.

Listing 8-1. The Contents of the placeOrder.html File

```
<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>

<div class="well">
    <h3>Ship to</h3>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" ng-model="data.shipping.name" />
    </div>

    <h3>Address</h3>

    <div class="form-group">
        <label>Street Address</label>
        <input class="form-control" ng-model="data.shipping.street" />
    </div>

    <div class="text-center">
        <button class="btn btn-primary">Complete order</button>
    </div>
</div>
```

The first thing to notice about this view is that I have not used the `ng-controller` directive to specify a controller. That means the view will be supported by the top-level controller, `sportsStoreCrtl`, which manages the view that contains the `ng-view` directive (which I introduced in Chapter 7). I make this point because you don't *have* to define controllers for partial views, which is convenient when the view doesn't require any additional behaviors, as is the case here.

The important AngularJS feature in the listing is the use of the `ng-model` directive on the `input` elements, like this:

```
...
<input class="form-control" ng-model="data.shipping.name" />
...
```

The `ng-model` directive sets up a *two-way data binding*. I explain data bindings in depth in Chapter 10, but the short version is that the kind of data binding I have been using so far in the SportsStore application—the ones that use the `{{` and `}}` characters—are *one-way bindings*, which means they simply display a value from the scope. The value a one-way binding displays can be filtered, or it can be an expression rather than just a data value, but it a read-only relationship. The value displayed by the binding will be updated if the corresponding value on the scope changes, but that's the only direction that updates flow in—from the scope to the binding.

Two-way data bindings are used on form elements to allow the user to enter values that change the scope, rather than just displaying them. Updates flow in both directions between the scope and the data binding. An update to the scope data property performed through a

JavaScript function, for example, will cause an `input` element to display the new value, and a new value entered by the user into the `input` element will update the scope. I explain the use of the `ng-model` directive in Chapter 10 and the broader AngularJS support for forms in Chapter 12. For this chapter it is enough to know that when the user enters a value into an `input` element, that value is assigned to the scope property specified by the `ng-model` directive—either the `data.shipping.name` property or the `data.shipping.street` property in this example. You can see how the form looks in the browser in Figure 8-1.

Tip Notice that I don't have to update the controller so that it defines a `data.shipping` object on its scope or the individual `name` or `street` properties. AngularJS scopes are remarkably flexible and assume that you want to define a property dynamically if it isn't already defined. I explain this in more detail in Chapter 13.

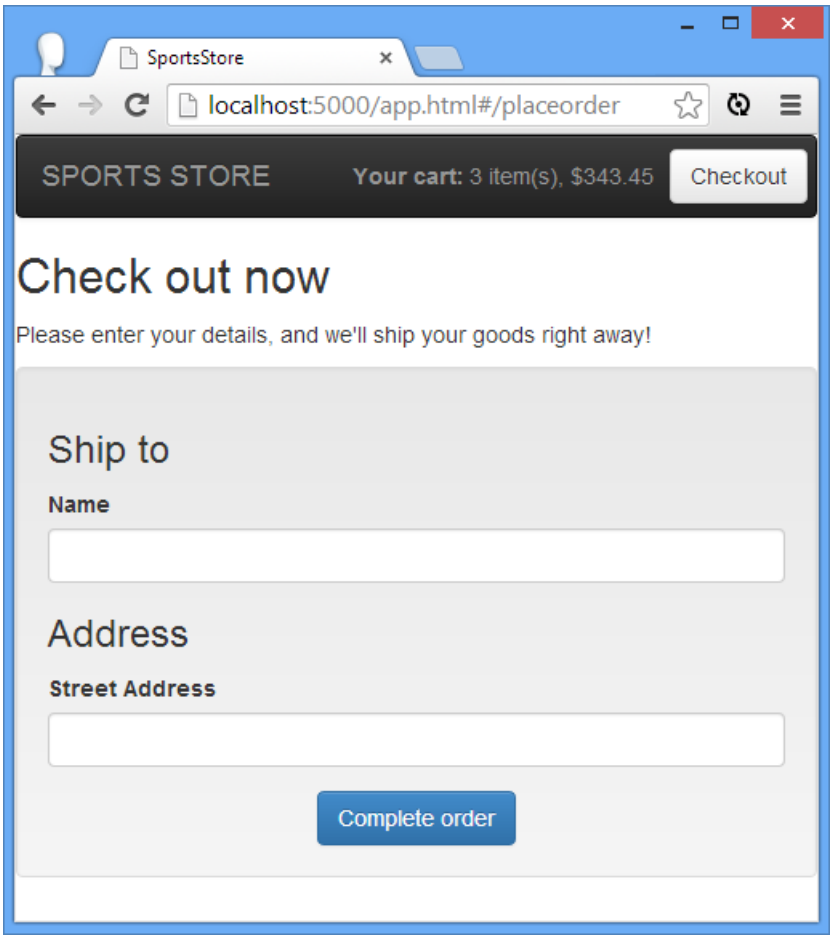

Figure 8-1. The short version of the shipping details form

Adding Form Validation

If you have written any kind of web application that uses form elements, then you will already know that users will put just about anything in an `input` field and that it is unwise to assume that users will have provided meaningful and useful data. To ensure you get the data you expect, AngularJS supports *form validation*, which allows values to be checked for suitability.

AngularJS form validation is based on honoring standard HTML attributes applied to form elements, such as `type` and `required`. Form validation is performed automatically, but some work is required to display validation feedback to the user and to integrate the overall validation results into an application.

Tip HTML5 defined a new set of values for the `type` attribute on `input` elements, which can be used to specify that a value should be an e-mail address or a number, for example. As I explain in Chapter 12, AngularJS can validate some of these new values.

Preparing for Validation

The first step in setting up form validation is to add a `form` element to the view and add the validation attributes to my `input` elements. Listing 8-2 shows the changes to the `placeOrder.html` file.

Listing 8-2. Preparing the placeOrder.html File for Validation

```
<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>

<form name="shippingForm" novalidate>
    <div class="well">
        <h3>Ship to</h3>
        <div class="form-group">
            <label>Name</label>
            <input class="form-control" ng-model="data.shipping.name" required />
        </div>

        <h3>Address</h3>

        <div class="form-group">
            <label>Street Address</label>
            <input class="form-control" ng-model="data.shipping.street" required />
        </div>

        <div class="text-center">
            <button class="btn btn-primary">Complete order</button>
        </div>
    </div>
</form>
```

The `form` element has three purposes, even though I won't be using the browser's built-in support for submitting forms in the SportsStore application.

The first purpose is to enable validation. AngularJS redefines some HTML elements with custom directives to enable special features, and one such element is `form`. Without a `form` element, AngularJS won't validate the contents of elements such as `input`, `select`, `textarea`, and so on.

The second purpose of the `form` element is to disable any validation that the browser might try to perform, which is done through the application of the `novalidate` attribute. This attribute is a standard HTML5 feature, and it ensures that only AngularJS is checking the data that the user provides. If you omit the `novalidate` attribute, then the user may get conflicting or duplicated validation feedback, depending on the browser being used.

The final purpose of the `form` element is to define a variable that will be used to report on the form validity. This is done through the `name` attribute, which I have set to `shippingForm`. You'll see how this value is used later in this chapter when I display validation feedback and when I wire up the `button` element so that the user can place the order only when the contents of the form are valid.

In addition to the `form` element, I have applied the `required` attribute to the `input` elements. This is one of the simplest validation attributes that AngularJS recognizes, and it means that the user has to provide a value—any value—for the `input` element to be valid. See Chapter 12 for details of the other ways in which you can validate form elements.

Displaying Validation Feedback

Once the `form` element and the validation attributes are in place, AngularJS starts to validate the data that the user provides, but I have to do a little more work to give the user any feedback. I get into the details in Chapter 12, but there are two kinds of feedback I can use: I can define CSS styles to take advantage of classes that AngularJS assigns valid and invalid form elements to, and I can use scope variables to control the visibility of targeted feedback messages for specific elements. Listing 8-3 shows both kinds of changes.

Listing 8-3. Applying Validation Feedback to the placeOrder.html File

```
<style>
    .ng-invalid { background-color: lightpink; }
    .ng-valid { background-color: lightgreen; }
    span.error { color: red; font-weight: bold; }
</style>

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>

<form name="shippingForm" novalidate>
    <div class="well">
        <h3>Ship to</h3>
        <div class="form-group">
            <label>Name</label>
            <input name="name" class="form-control"
                ng-model="data.shipping.name" required />
            <span class="error" ng-show="shippingForm.name.$error.required">
```

```
                Please enter a name
            </span>
        </div>

        <h3>Address</h3>

        <div class="form-group">
            <label>Street Address</label>
            <input name="street" class="form-control"
                ng-model="data.shipping.street" required />
            <span class="error" ng-show="shippingForm.street.$error.required">
                Please enter a street address
            </span>
        </div>

        <div class="text-center">
            <button class="btn btn-primary">Complete order</button>
        </div>
    </div>
</form>
```

AngularJS assigns form elements to the `ng-valid` and `ng-invalid` classes, so I started by defining a `style` element that contains CSS styles that target those classes. Form elements are always in one of these classes, such that one of these styles is always applied.

Tip I am setting up a simple validation configuration for the SportsStore application, the effect of which is that the form is invalid from the moment that it is shown to the user. This isn't always acceptable, and in Chapter 12 I describe some additional features that AngularJS provides to control when validation messages are displayed.

The CSS styles have the effect of indicating when there is a problem with an `input` element but provide no indication what the problem is. For that I have to add a `name` attribute to each element and use some validation data that AngularJS adds to the scope to control the visibility of error messages, like this:

```
...
<input name="street" class="form-control" ng-model="data.shipping.street" required />
<span class="error" ng-show="shippingForm.street.$error.required">
    Please enter a street address
</span>
...
```

In this fragment, I have shown the `input` element that captures the user's street address, which I have assigned the `name` value of `street`. AngularJS creates a `shippingForm.street` object on the scope (which is the combination of the `name` of the `form` element and the `name` of the `input` element). This object defines a `$error` property, which itself is an object that has properties for each of the validation attributes that the contents of the `input` element fail to satisfy. Or, to put it another way, if the `shippingForm.street.$error.required` property is `true`, then I know that the contents of the `street input` element are invalid, which I use to display an error message to the user through the application of the `ng-show` directive. (I explain the validation properties fully in

Chapter 12 and the `ng-show` directive in Chapter 11.) You can see the initial state of the form in Figure 8-2.

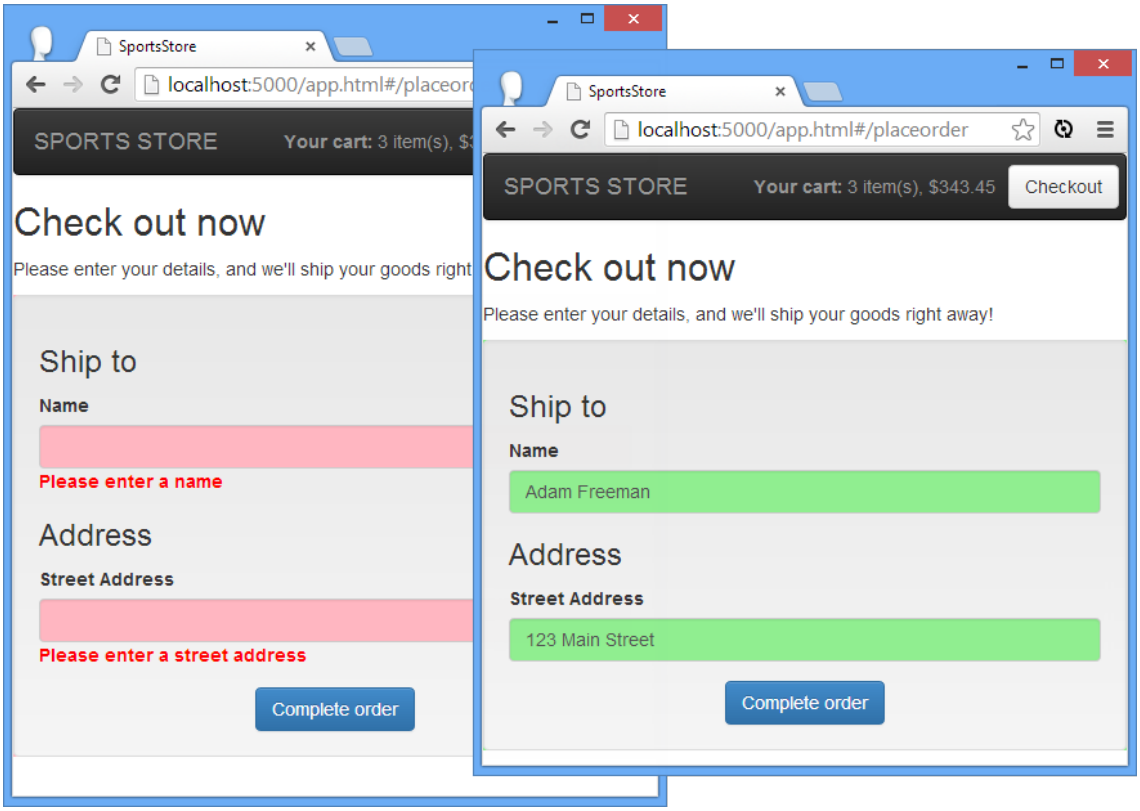

Figure 8-2. The initial (invalid) form

I satisfy the `required` attribute as I enter details into the `input` elements, which has the effect of switching the color applied to the element from red to green and hiding the error message.

Note I am deliberately simplifying the way I apply validation in this chapter, but AngularJS can be used to create much more subtle and pleasing validation configurations, as I describe in Chapter 12.

Linking the Button to Validity

In most web applications, the user shouldn't be able to move to the next step in a process until all the form data has been provided and is valid. To that end, I want to disable the Complete order button when the form is invalid and automatically enable it when the user has completed the form properly.

To do this, I can take advantage of the validation information that AngularJS adds to the scope. In addition to the per-field information that I used in the previous section to display per-

element messages, I can get information about the overall state of the form as well. The `shippingForm.$invalid` property will be set to `true` when one or more of the `input` elements is invalid, and I can combine this with the `ng-disabled` directive to manage the state of the `button` element. I describe the `ng-disabled` directive in Chapter 11, but it adds and removes the `disabled` attribute from the element it has been applied to based on the scope property or expression it is configured with. Listing 8-4 shows how I can tie the state of the `button` to form validation.

Listing 8-4. Setting the State of the Button in the placeOrder.html File

```
...
<div class="text-center">
    <button ng-disabled="shippingForm.$invalid"
        class="btn btn-primary">Complete order</button>
</div>
...
```

You can see the effect that the `ng-disabled` directive has on the `button` element in Figure 8-3.

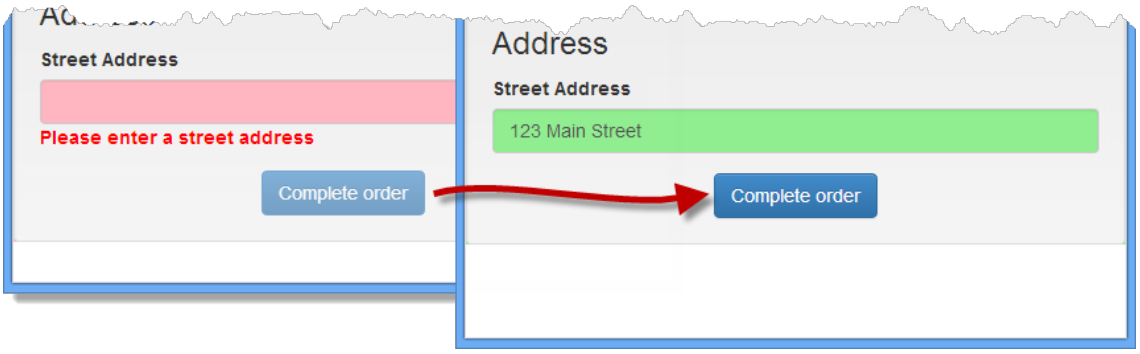

Figure 8-3. Controlling the state of a button based on form validation

Adding the Remaining Form Fields

Now that you have seen how AngularJS form validation works, I am going to add the remaining `input` elements to the form. I avoided this earlier because I wanted to show you the individual validation features without listing duplicate markup, but I can't go any further without the completed form. Listing 8-5 shows the addition of the remaining `input` elements and their associated validation messages.

Listing 8-5. Adding the Remaining Form Fields to the placeOrder.html File

```
<style>
    .ng-invalid { background-color: lightpink; }
    .ng-valid { background-color: lightgreen; }
    span.error { color: red; font-weight: bold; }
</style>

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>
```

```
<form name="shippingForm" novalidate>
    <div class="well">
        <h3>Ship to</h3>
        <div class="form-group">
            <label>Name</label>
            <input name="name" class="form-control"
                ng-model="data.shipping.name" required />
            <span class="error" ng-show="shippingForm.name.$error.required">
                Please enter a name
            </span>

        </div>

        <h3>Address</h3>

        <div class="form-group">
            <label>Street Address</label>
            <input name="street" class="form-control"
                ng-model="data.shipping.street" required />
            <span class="error" ng-show="shippingForm.street.$error.required">
                Please enter a street address
            </span>
        </div>

        <div class="form-group">
            <label>City</label>
            <input name="city" class="form-control"
                ng-model="data.shipping.city" required />
            <span class="error" ng-show="shippingForm.city.$error.required">
                Please enter a city
            </span>
        </div>

        <div class="form-group">
            <label>State</label>
            <input name="state" class="form-control"
                ng-model="data.shipping.state" required />
            <span class="error" ng-show="shippingForm.state.$error.required">
                Please enter a state
            </span>
        </div>

        <div class="form-group">
            <label>Zip</label>
            <input name="zip" class="form-control"
                ng-model="data.shipping.zip" required />
            <span class="error" ng-show="shippingForm.zip.$error.required">
                Please enter a zip code
            </span>
        </div>

        <div class="form-group">
            <label>Country</label>
            <input name="country" class="form-control"
                ng-model="data.shipping.country" required />
            <span class="error" ng-show="shippingForm.country.$error.required">
```

```
                Please enter a country
            </span>
        </div>

        <h3>Options</h3>
        <div class="checkbox">
            <label>
                <input name="giftwrap" type="checkbox"
                    ng-model="data.shipping.giftwrap" />
                Gift wrap these items
            </label>
        </div>

        <div class="text-center">
            <button ng-disabled="shippingForm.$invalid"
                    class="btn btn-primary">Complete order</button>
        </div>
    </div>
</form>
```

Tip The markup shown in Listing 8-5 is highly duplicative and is the sort of thing that attracts typos. You might be tempted to try to use the `ng-repeat` directive to generate the `input` elements from an array of objects that describes each field. This doesn't work well because of the way that the attribute values for directives like `ng-model` and `ng-show` are evaluated within the scope of the `ng-repeat` directive. My advice is to simply accept the duplication in the markup, but if you do want a more elegant technique, then read Chapters 15–17, which describe the ways in which you can create custom directives.

Placing Orders

Even though the state of the `button` element is controlled by form validation, clicking the `button` has no effect, and that's because I need to finish off the SportsStore application by allowing the user to submit orders. In the sections that follow, I'll extend the database provided by Parse.com, send order data to the server using an Ajax request, and display a final thank-you message to complete the process.

Extending the Backend

I need to extend the Parse.com configuration to capture the orders that the SportsStore application will submit. Using the Parse.com Data Browser (which I first used in Chapter 6), click the New Class button and select the Custom option and enter `Orders` into the text field, as shown in Figure 8-4.

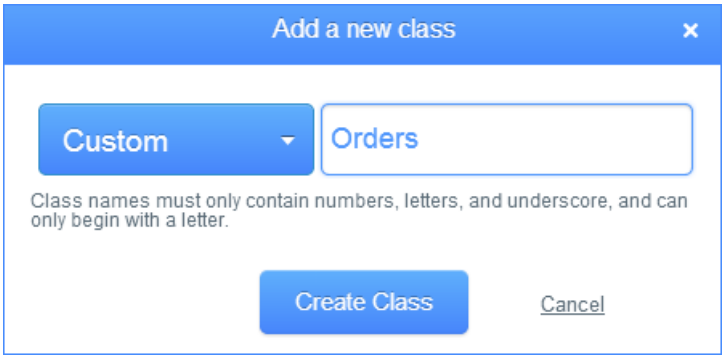

Figure 8-4. Creating the Orders Class

Set the name of the new collection to `/orders` and click the `Create` button. The Parse.com Data Browser will display a new data grid. Click the `+Col` button and create the columns shown in Table 8-1.

Table 8-1. *The Properties Required for the Orders Collection*

Name	Type
`name`	`String`
`street`	`String`
`city`	`String`
`state`	`String`
`zip`	`String`
`country`	`String`
`giftwrap`	`Boolean`
`products`	`Array`

Pay particular attention to the type of the `giftwrap` and `products` properties—they are not the same type as the other properties, and you'll get some odd results if you don't define them correctly.

Defining the Controller Behavior

The next step is to define the controller behavior that will send details of an order to the server using an Ajax request. I could define this functionality in a number of different ways—as a

service or in a new controller, for example. This flexibility is one of the hallmarks of working with AngularJS. There is no absolute right or wrong when it comes to the structure of an AngularJS application, and you will develop your own style and set of preferences as your experience builds. I am going to keep things simple and add the behavior I need to the top-level `sportsStore` controller, which already contains the code that makes the Ajax request to load the product data. Listing 8-6 shows the changes I have made.

Listing 8-6. Sending the Order to the Server in the sportsStore.js File

```
angular.module("sportsStore")
    .constant("dataUrl", "https://api.parse.com/1/classes/Products")
    .constant("orderUrl", "https://api.parse.com/1/classes/Orders")
    .run(function ($http) {
        $http.defaults.headers.common["X-Parse-Application-Id"]
            = "McOi2Q6FZNovOr4bQD6QRFwzgw4iRHOy2JcDYuLq";
        $http.defaults.headers.common["X-Parse-REST-API-Key"]
            = "h4dnGScEtCWMMjaWmCJYNBZrDCrnROZmsKvEwXLD";
    })
    .controller("sportsStoreCtrl", function ($scope, $http, $location,
        dataUrl, orderUrl, cart) {

        $scope.data = {
        };

        $http.get(dataUrl)
            .success(function (data) {
                $scope.data.products = data.results;
            })
            .error(function (response) {
                $scope.data.error = response.error || response;
            });

        $scope.sendOrder = function (shippingDetails) {
            var order = angular.copy(shippingDetails);
            order.products = cart.getProducts();
            $http.post(orderUrl, order)
                .success(function (data) {
                    $scope.data.orderId = data.objectId;
                    cart.getProducts().length = 0;
                })
                .error(function (error) {
                    $scope.data.orderError = error;
                }).finally(function () {
                    $location.path("/complete");
                });
        }
    });
```

Parse.com will create a new object in the database in response to a POST request and will return the object that it has created in the response, including the `objectId` attribute that has been generated to reference the new object.

Knowing this, you can see how the new additions to the controller operate. I have defined a new `constant` that specifies the URL that I will use for the POST request and added a dependency for the `cart` service so that I can get details of the products that the user requires.

The behavior I added to the controller is called `sendOrder`, and it receives the shipping details for the user as its argument.

I use the `angular.copy` utility method, which I describe in Chapter 5, to create a copy of the shipping details object so that I can safely manipulate it without affecting other parts of the application. The properties of the shipping details object—which are created by the `ng-model` directives in the previous section—correspond to the properties that I defined for the `Orders` class, and all I have to do is define a `products` property that references the array of products in the cart.

I use the `$http.post` method, which creates an Ajax POST request to the specified URL and data, and I use the `success` and `error` methods that I introduced in Chapter 5 (and which I describe fully in Chapter 20) to respond to the outcomes from the request. For a successful request, I assign the `objectId` of the newly created order object to a scope property and clear the contents of the cart. If there is a problem, I assign the error object to the scope so that I can refer to it later.

I also use the `then` method on the promise returned by the `$http.post` method. The `then` method takes a function that is invoked whatever the outcome of the Ajax request. I want to display the same view to the user whatever happens, so I use the `then` method to call the `$location.path` method. This is how the path component of the URL is set programmatically, and it will trigger a change of view through the URL configuration that I created in Chapter 7. (I describe the `$location` service in Chapter 11 and demonstrate its use with URL routing in Chapter 22.)

Calling the Controller Behavior

To invoke the new controller behavior, I need to add the `ng-click` directive to the button element in the shipping details view, as shown in Listing 8-7.

Listing 8-7. Adding a Directive to the placeOrder.html File

```
...
<div class="text-center">
    <button ng-disabled="shippingForm.$invalid"
            ng-click="sendOrder(data.shipping)"
            class="btn btn-primary">
        Complete order
    </button>
</div>
...
```

Defining the View

The URL path that I specify after the Ajax request has completed is `/complete`, which the URL routing configuration maps to the file `/views/thankYou.html`. I created this file, and you can see the contents of it in Listing 8-8.

Listing 8-8. The Contents of the thankYou.html File

```
<div class="alert alert-danger" ng-show="data.orderError">
```

```
    Error ({{data.orderError.status}}). The order could not be placed.
    <a href="#/placeorder" class="alert-link">Click here to try again</a>
</div>

<div class="well" ng-hide="data.orderError">
    <h2>Thanks!</h2>
    Thanks for placing your order. We'll ship your goods as soon as possible.
    If you need to contact us, use reference {{data.orderId}}.
</div>
```

This view defines two different blocks of content to deal with success and unsuccessful Ajax requests. If there has been an error, then details of the error are displayed, along with a link that takes the user back to the shipping details view so they can try again. The user is shown a thank-you message that contains the `objectId` of the new order object if the request is successful. You can see the successful outcome in Figure 8-5.

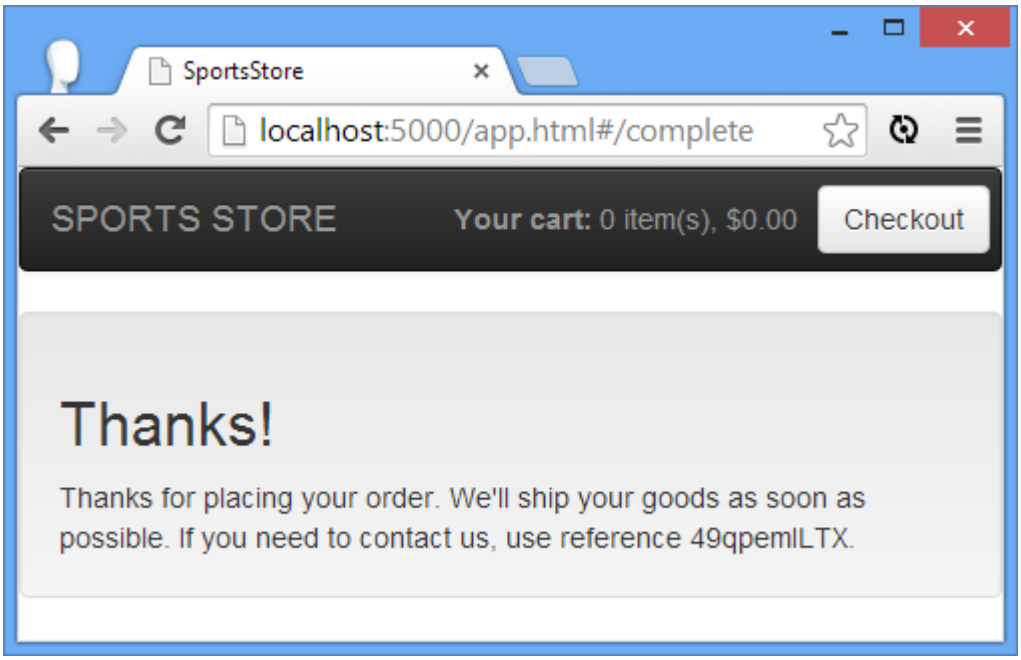

Figure 8-5. Displaying feedback to the user when an order is placed

Making Improvements

In building the user side of the SportsStore application, I took a couple of shortcuts that could be improved upon with techniques that I describe in later chapters but that depend on some concepts that I didn't want to introduce here.

First, when you load the `app.html` file into the browser, you may notice a small delay between the view being displayed and the elements for the products and categories being generated. This is because the Ajax request that gets the data is happening in the background, and while waiting for the server to return the data, AngularJS carries on executing the

application and displaying the views, which are then updated when the data arrives. In Chapter 22, I describe how you can use the URL routing feature to prevent AngularJS from displaying the view until the Ajax request has been completed.

Next, I process the product data to extract the set of categories for the navigation and pagination features. In a real project, I would consider generating this information once when the product data first arrives and then reusing it thereafter. In Chapter 20, I describe how you can use promises to build chains of behavior, which is ideally suited to this kind of task.

Finally, I would have used the `$animate` service, which I describe in Chapter 23, to display short, focused animations to ease the transition from one view to another when the URL path changes.

AVOIDING OPTIMIZATION PITFALLS

You will notice that I say that I could *consider* reusing the category and pagination data, not that I would definitely do so. That's because any kind of optimization should be carefully assessed to ensure it is sensible and that it avoids two main pitfalls that dog optimization efforts.

The first pitfall is *premature optimization*, which is where a developer sees an opportunity to optimize an operation or task before the current implementation causes any problems or breaks a contract in the nonfunctional specification. This kind of optimization tends to make code more specific in its nature that it would otherwise be, and that can kill the easy movement of functionality from one component to another that is typical of AngularJS (and is one of the most enjoyable aspects of AngularJS development). Further, by optimizing code that hasn't been identified as a problem, you are spending time solving a (*potential*) problem that no one cares about—time that could equally be spent fixing real problems or building features that users require.

The second pitfall is *translation optimization*, where the optimization simply changes the nature of the problem rather than offers a real solution. The main issue with the way that the category and pagination data is generated is that it requires computation that could be avoided by caching the information. This seems like a good idea, but caching requires memory, which is often in short supply in mobile devices. The same kinds of devices that would benefit from not having to process a few data records are the same ones that lack the capacity to store some additional data to avoid that computation. And, if you are sending the client so much data that the user has to wait while the processing is performed, then the problems are more fundamental, and you should consider the way you have designed your application—perhaps obtaining and processing data in smaller chunks would be a more sensible solution.

I am not saying that you should not optimize your applications, but I *am* saying that you should not do so until you have a real problem to solve and that your optimizations should be a solution to the problem. Don't let an abhorrence of inefficiency prevent you from seeing that your development time is important and should only be spent solving real issues.

Administering the Product Catalog

To complete the SportsStore application, I am going to create an application that will allow the administrator to manage the contents of the product catalog and the order queue. This will allow me to demonstrate how AngularJS can be used to perform *create*, *read*, *update*, and *delete*

(CRUD) operations and reinforce the use of some key features from the main SportsStore application.

Note Every back-end service implements authentication in a slightly different way, but the basic premise is the same: Send a request with the user's credentials to a specific URL, and if the request is successful, the browser will return a cookie that the browser will automatically send with subsequent requests to identify the user. The examples in this section are specific to Parse.com, but they will translate easily to most platforms.

Preparing Parse.com

Making changes to the database is something that only administrators should be able to do. To that end, I am going to use Parse.com to define an administrator user and create the access policy described by Table 8-2.

Table 8-2. *The Access Control Policy*

Class	Admin	User
`products`	Get, Find, Update, Create, Delete, Add Fields	Get, Find
`orders`	Get, Find, Update, Create, Delete, Add Fields	Create

In short, the administrator should be able to perform any operation on any class of object. The normal users should be able to read (but not modify) the `products` collection and create new objects in the `orders` collection (but not be able to see, modify, or delete them).

Go to the Parse.com Data Browser and click the New Class button. Select the User option, as shown in Figure 8-6, and click the Create Class button.

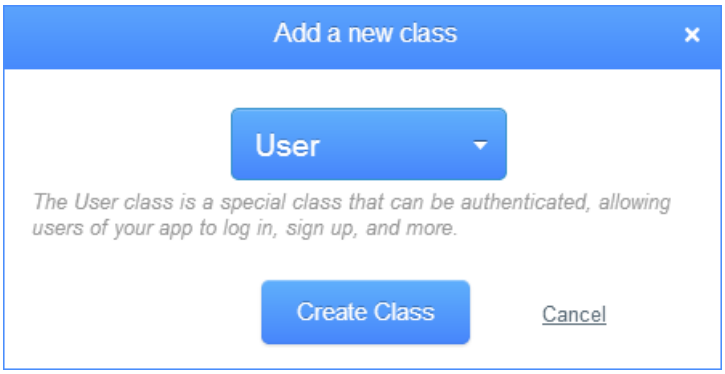

Figure 8- 6. Creating the User Class

Click the `+Row` button and enter the data shown into Table 803 into the appropriate columns. You can ignore the other columns – they are not required for the SportsStore application.

Table 8-4. *Columns Details for the Administration User*

Column	Value
username	admin
password	secret

When you create the new row, Parse.com will assign a new `objectId` value to the user object. Make a note of this value, which you will need in the following section. My `objectId` value is `kcpcwsyST5` but you will have a different value.

Securing the Object Classes

Select the `Orders` collection in the Classes list of the Parse.com Data Browser, click the `More` button and select `Set Permissions`. Click on all of the actions shown in the list except `Create`, for which the default values are fine, uncheck the `Any User Can Perform This Action` option and enter the `objectId` value from the previous section into the second text field, as shown in Figure 8-7. (Do not use the first text field, which is for roles. There is only one user in this example, so I am not going to go to the trouble of creating security roles).

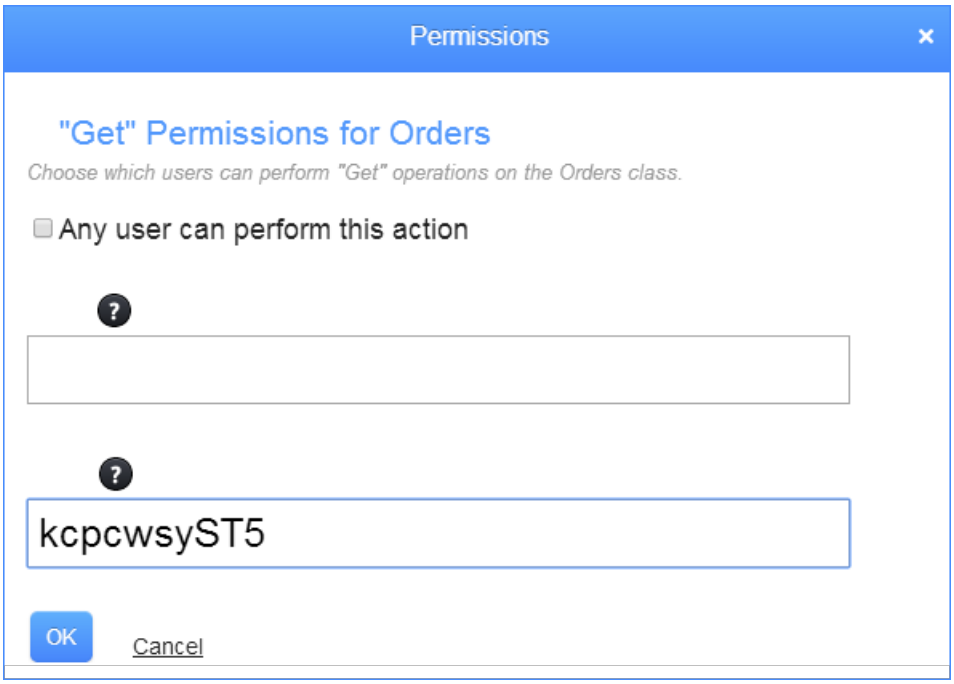

Figure 8-7. Changing the Permissions for an Action

When you have set all of the individual actions, the overall settings should like those shown in Figure 8-8.

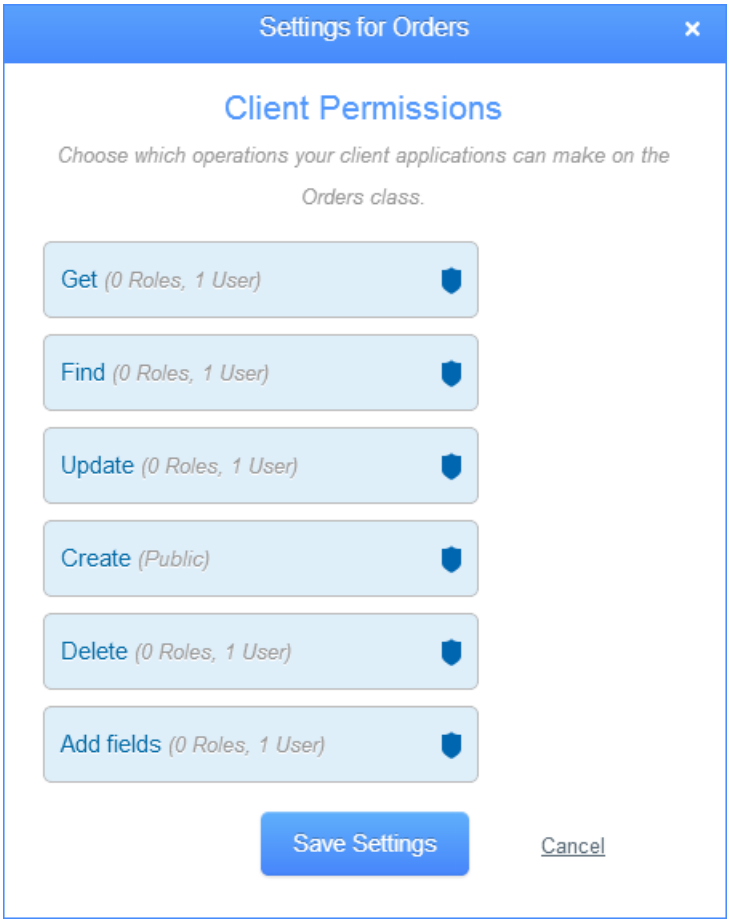

Figure 8-8. Setting the Security Policy for the Orders Class

Repeat the process for all of the actions on the `Products` class except `Get` and `Find` as shown in Figure 8-9.

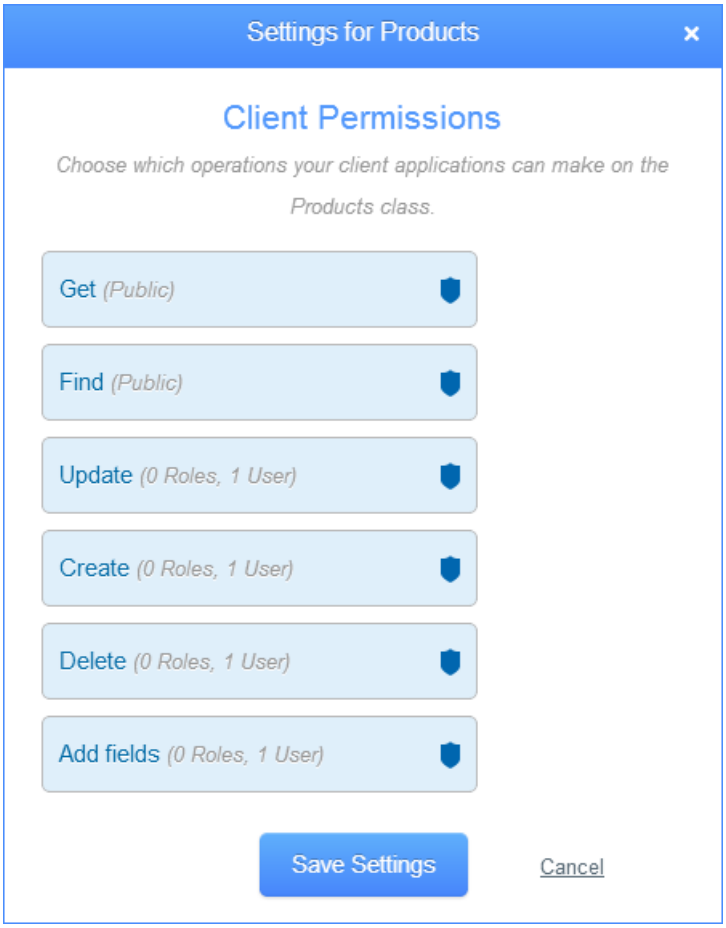

Figure 8-9. Setting the Security Policy for the Orders Class

Creating the Admin Application

I am going to create a separate AngularJS application for the administration tasks. I could integrate these features into the main application, but that would mean all users would be required to download the code for the admin functions, even though most of them would never use it. I added a new file called `admin.html` to the `angularjs` folder, the contents of which are shown in Listing 8-9.

Listing 8-9 .The Contents of the admin.html File

```
<!DOCTYPE html>
<html ng-app="sportsStoreAdmin">
<head>
    <title>Administration</title>
    <script src="angular.js"></script>
    <script src="ngmodules/angular-route.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
```

```
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStoreAdmin", ["ngRoute"])
            .config(function ($routeProvider) {

                $routeProvider.when("/login", {
                    templateUrl: "/views/adminLogin.html"
                });

                $routeProvider.when("/main", {
                    templateUrl: "/views/adminMain.html"
                });

                $routeProvider.otherwise({
                    redirectTo: "/login"
                });
            });
    </script>
</head>
<body>
    <ng-view />
</body>
</html>
```

This HTML file contains the `script` and `link` elements required for the AngularJS and Bootstrap files and an inline `script` element that defines the `sportsStoreAdmin` module, which will contain the application functionality (and which I have applied to the `html` element using the `ng-app` directive). I have used the `Module.config` method to create three routes for the application, which drive the `ng-view` directive in the `body` element. Table 8-5 summarizes the paths that the URLs match and the view files that they load.

Table 8-5. *The URL Paths in the admin.html File*

URL Path	View
`/login`	`/views/adminLogin.html`
`/main`	`/views/adminMain.html`
All others	Redirects to `/login`

For the route defined with the `otherwise` method, I used the `redirectTo` option, which changes the URL path to another route. This has the effect of moving the browser to the `/login` path, which is the one that I will use to authenticate the user. I describe the complete set of configuration options that you can use with URL routes in Chapter 22.

Adding the Placeholder View

I am going to implement the authentication feature first, but I need to create some placeholder content for the `/views/adminMain.html` view file so that I have something to show when authentication is successful. Listing 8-10 shows the (temporary) contents of the file.

Listing 8-10. The Contents of the adminMain.html File

```
<div class="well">
    This is the main view
</div>
```

I'll replace this placeholder with useful content once the application is able to authenticate users.

Implementing Authentication

Parse.com authenticates users using standard HTTP requests. The application sends a GET request to the `/1/login` URL, which includes `username` and `password` values for the authenticating user. The server responds with status code `200` if the authentication attempt is successful and code `401` or `404` when the user cannot be authenticated. To implement authentication, I started by defining a controller that makes the Ajax calls and deals with the response. Listing 8-11 shows the contents of the `controllers/adminControllers.js` file, which I created for this purpose.

Listing 8-11. The Contents of the adminControllers.js File

```
angular.module("sportsStoreAdmin")
    .constant("authUrl", "https://api.parse.com/1/login")
    .run(function ($http) {
        $http.defaults.headers.common["X-Parse-Application-Id"]
            = "McOi2Q6FZNovOr4bQD6QRFwzgw4iRHOy2JcDYuLq";
        $http.defaults.headers.common["X-Parse-REST-API-Key"]
            = "h4dnGScEtCWMMjaWmCJYNBZrDCrnROZmsKvEwXLD";
    })

.controller("authCtrl", function ($scope, $http, $location, authUrl) {

    $scope.authenticate = function (user, pass) {
        $http.get(authUrl, {
            params: {
                username: user,
                password: pass
            },
        }).success(function (data) {
            $scope.$broadcast("sessionToken", data.sessionToken);
            $http.defaults.headers.common["X-Parse-Session-Token"]
                = data.sessionToken;
            $location.path("/main");
        }).error(function (response) {
            $scope.authenticationError = response.error || response;
        });
    }
});
```

I use the `angular.module` method to extend the `sportsStoreAdmin` module that is created in the `admin.html` file. I use the `constant` method to specify the URL that will be used for authentication and the run method to define the key headers required to use the `Parse.com` web service.

I have created an `authCtrl` controller that defines a behavior called `authenticate` that receives the `username` and `password` values as arguments and makes an Ajax request to parse.com with the

$http.get method (which I describe in Chapter 20). I use the $location service, which I describe in Chapter 11, to programmatically change the path displayed by the browser (and so trigger a URL route change) if the Ajax request is successful.

If the server returns an error, I assign the message passed to the error function to a scope variable so that I can display details of the problem to the user. If the authentication request is successful, then Parse.com returns an object with a sessionToken property whose value must be included in subsequent requests using a header called X-Parse-Session-Token.

I will need to use the value of the sessionToken property to set the X-Parse-Session-Token header for the other HTTP requests I need to administer the application and so I have used the $scope.$broadcast method to send an event that contains the token value. You will see how I receive the event with the $scope.$on method in Listing 8-20. Events are the means by which controllers can communicate and I describe them in Chapter 13.

I need to include the JavaScript file that contains the controller in the admin.html file, taking care to ensure that it appears after the script element that defines the module that is being extended. Listing 8-12 shows the change to the admin.html file.

Listing 8-12. Adding a script Element for a Controller to the admin.html File

```
<!DOCTYPE html>
<html ng-app="sportsStoreAdmin">
<head>
    <title>Administration</title>
    <script src="angular.js"></script>
    <script src="ngmodules/angular-route.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStoreAdmin", ["ngRoute"])
            .config(function ($routeProvider) {

                $routeProvider.when("/login", {
                    templateUrl: "/views/adminLogin.html"
                });

                $routeProvider.when("/main", {
                    templateUrl: "/views/adminMain.html"
                });

                $routeProvider.otherwise({
                    redirectTo: "/login"
                });
            });
    </script>
    <script src="controllers/adminControllers.js"></script>
</head>
<body>
    <ng-view />
</body>
</html>
```

Defining the Authentication View

The next step is to create the view that will allow the user to enter a username and password, invoke the `authenticate` behavior defined by the `authCtrl` controller, and display details of any errors. Listing 8-13 shows the contents of the `views/adminLogin.html` file.

Listing 8-13. The Contents of the adminLogin.html File

```
<div class="well" ng-controller="authCtrl">

    <div class="alert alert-info" ng-hide="authenticationError">
        Enter your username and password and click Log In to authenticate
    </div>

    <div class="alert alert-danger" ng-show="authenticationError">
        Authentication Failed ({{authenticationError}}). Try again.
    </div>

    <form name="authForm" novalidate>
        <div class="form-group">
            <label>Username</label>
            <input name="username" class="form-control" ng-model="username" required />
        </div>
        <div class="form-group">
            <label>Password</label>
            <input name="password" type="password" class="form-control"
                   ng-model="password" required />
        </div>
        <div class="text-center">
            <button ng-click="authenticate(username, password)"
                    ng-disabled="authForm.$invalid"
                    class="btn btn-primary">
                Log In
            </button>
        </div>
    </form>
</div>
```

This view uses techniques I introduced for the main SportsStore application and that I describe in depth in later chapters. I use the `ng-controller` directive to associate the view with the `authCtrl` controller. I use the AngularJS support for forms and validation (Chapter 12) to capture the details from the user and prevent the Log In button from being clicked until values for both the username and password have been entered. I use the `ng-model` directive (Chapter 10) to assign the values entered to the scope. I use the `ng-show` and `ng-hide` directives (Chapter 11) to prompt the user to enter credentials and to report on an error. Finally, I use the `ng-click` directive (Chapter 11) to invoke the `authenticate` behavior on the controller to perform authentication.

You can see how the view is displayed by the browser in Figure 8-10. To authenticate, enter the username (`admin`) and password (`secret`) that Parse.com is expecting and click the button.

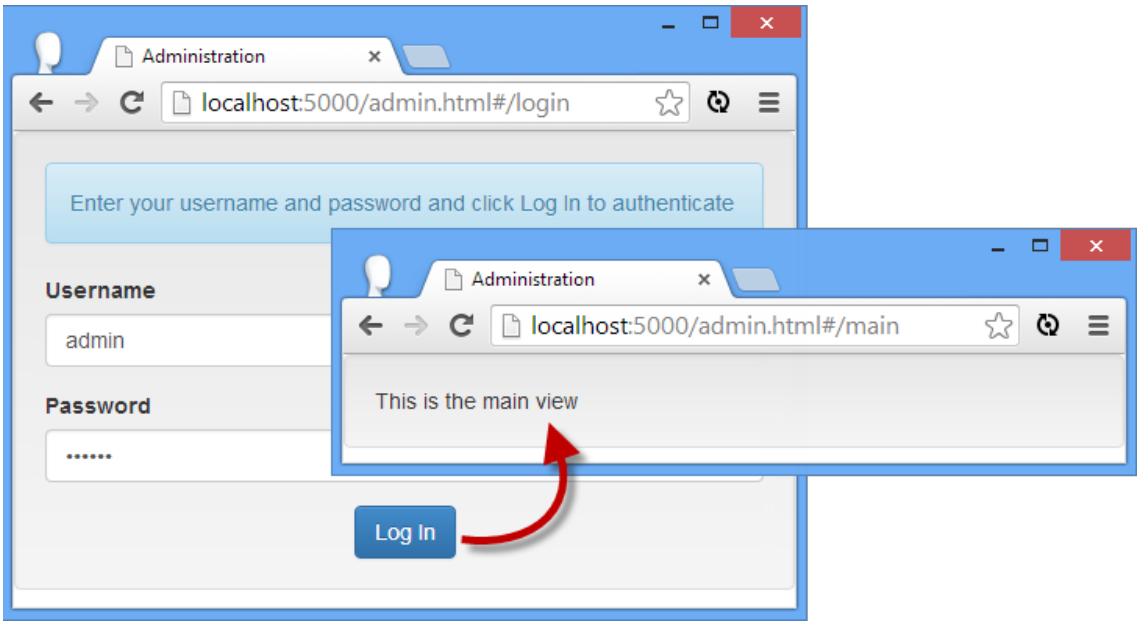

Figure 8-10. Authenticating the user

Defining the Main View and Controller

Once the user is authenticated, the `ng-view` directive displays the `adminMain.html` view. This view will be responsible for allowing the administrator to manage the contents of the product catalog and see the queue of orders.

Before I start to define the functionality that will drive the application, I need to define placeholder content for the views that will display the list of products and orders. First, I created `views/adminProducts.html`, the content of which is shown in Listing 8-14.

Listing 8-14. The Contents of the adminProducts.html File

```
<div class="well">
    This is the product view
</div>
```

Next, I create the `views/adminOrders.html` file, for which I have defined a similar placeholder, as shown in Listing 8-15.

Listing 8-15. The Contents of the adminOrders.html File

```
<div class="well">
    This is the order view
</div>
```

I need the placeholders so I can demonstrate the flow of views in the admin application. The URL routing feature has a serious limitation: You can't nest multiple instances of the `ng-view` directive, which makes it slightly more difficult to arrange to display different views within the

scope of ng-view. I am going to demonstrate how to address this using the ng-include directive as a slightly less elegant—but perfectly functional—alternative. I started by defining a new controller in the adminControllers.js file, as shown in Listing 8-16.

Listing 8-16. Adding a New Controller in the adminControllers.js File

```
angular.module("sportsStoreAdmin")
    .constant("authUrl", "https://api.parse.com/1/login")
    .run(function ($http) {
        $http.defaults.headers.common["X-Parse-Application-Id"]
            = "McOi2Q6FZNovOr4bQD6QRFwzgw4iRHOy2JcDYuLq";
        $http.defaults.headers.common["X-Parse-REST-API-Key"]
            = "h4dnGScEtCWMMjaWmCJYNBZrDCrnROZmsKvEwXLD";
    })

.controller("authCtrl", function ($scope, $http, $location, authUrl) {

    $scope.authenticate = function (user, pass) {
        $http.get(authUrl, {
            params: {
                username: user,
                password: pass
            },
        }).success(function (data) {
            $scope.$broadcast("sessionToken", data.sessionToken);
            $http.defaults.headers.common["X-Parse-Session-Token"]
                = data.sessionToken;
            $location.path("/main");
        }).error(function (response) {
            $scope.authenticationError = response.error || response;
        });
    }
})
.controller("mainCtrl", function ($scope) {

    $scope.screens = ["Products", "Orders"];
    $scope.current = $scope.screens[0];

    $scope.setScreen = function (index) {
        $scope.current = $scope.screens[index];
    };

    $scope.getScreen = function () {
        return $scope.current == "Products"
            ? "/views/adminProducts.html" : "/views/adminOrders.html";
    };
});
```

The new controller is called mainCtrl, and it provides the behaviors and data I need to use the ng-include directive to manage views, as well as generate the navigation buttons that will switch between the views. The setScreen behavior is used to change the displayed view, which is exposed through the getScreen behavior.

You can see how the controller functionality is consumed in Listing 8-17, which shows how I have revised the adminMain.html file to remove the placeholder functionality.

Listing 8-17. Revising the adminMain.html File

```
<div class="panel panel-default row" ng-controller="mainCtrl">
    <div class="col-xs-3 panel-body">
        <a ng-repeat="item in screens" class="btn btn-block btn-default"
           ng-class="{'btn-primary': item == current }" ng-click="setScreen($index)">
            {{item}}
        </a>
    </div>
    <div class="col-xs-8 panel-body" >
        <div ng-include="getScreen()" />
    </div>
</div>
```

This view uses the `ng-repeat` directive to generate `a` elements for each value in the scope `screens` array. As I explain in Chapter 10, the `ng-repeat` directive defines some special variables that can be referred to within the elements it generates, and one of those, `$index`, returns the position of the current item in the array. I use this value with the `ng-click` directive, which invokes the `setScreen` controller behavior.

The most important part of this view is the use of the `ng-include` directive, which I introduced in Chapter 7 to display a single partial view and which I describe properly in Chapter 10. The `ng-include` directive can be passed a behavior that is invoked to obtain the name of the view that should be displayed, as follows:

```
...
<div ng-include="getScreen()" />
...
```

I have specified the `getScreen` behavior, which maps the currently selected navigation value to one of the views I defined at the start of this section. You can see the buttons that the `ng-repeat` directive generates—and the effect of clicking them—in Figure 8-11. This isn't as elegant or robust as using the URL routing feature, but it is functional and is a useful technique in complex applications where a single instance of the `ng-view` directive doesn't provide the depth of control over views that is required.

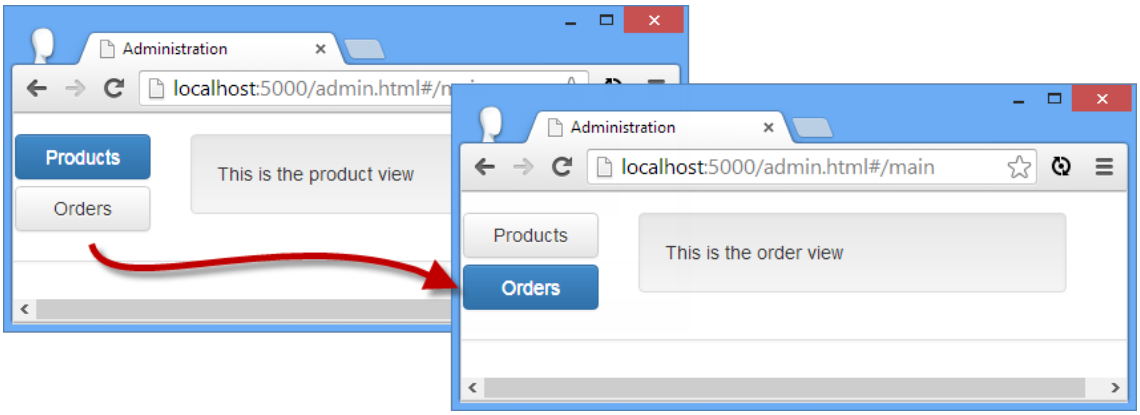

Figure 8-11. Using the ng-include directive to select views

Implementing the Orders Feature

I am going to start with the list of orders, which is the simplest to deal with because I am only going to display a read-only list. In a real e-commerce application, orders would go into a complex workflow that would involve payment validation, inventory management, picking and packing, and—ultimately—shipping the ordered products. As I explained in Chapter 6, these are not features you would implement using AngularJS, so I have omitted them from the SportsStore application. With that in mind, I have added a new controller to the `adminControllers.js` file that uses the `$http` service to make an Ajax GET request to Parse.com to get the orders, as shown in Listing 8-18.

Listing 8-18. Adding a Controller to Obtain the Orders in the adminControllers.js File

```
angular.module("sportsStoreAdmin")
    .constant("authUrl", "https://api.parse.com/1/login")
    .constant("ordersUrl", "https://api.parse.com/1/classes/Orders")
    .run(function ($http) {
        $http.defaults.headers.common["X-Parse-Application-Id"]
            = "McOi2Q6FZNovOr4bQD6QRFwzgw4iRHOy2JcDYuLq";
        $http.defaults.headers.common["X-Parse-REST-API-Key"]
            = "h4dnGScEtCWMMjaWmCJYNBZrDCrnROZmsKvEwXLD";
    })

.controller("authCtrl", function ($scope, $http, $location, authUrl) {

    // ...controller statements omitted for brevity...

})
.controller("mainCtrl", function ($scope) {

    // ...controller statements omitted for brevity...

})
.controller("ordersCtrl", function ($scope, $http, ordersUrl) {

    $http.get(ordersUrl)
        .success(function (data) {
            $scope.orders = data.results;
        })
        .error(function (response) {
            $scope.error = response.error || response;
        });

    $scope.selectedOrder;

    $scope.selectOrder = function (order) {
        $scope.selectedOrder = order;
    };

    $scope.calcTotal = function (order) {
        var total = 0;
        for (var i = 0; i < order.products.length; i++) {
            total +=
                order.products[i].count * order.products[i].price;
```

```
        }
        return total;
    }
});
```

I have defined a constant that defines the URL that will return a list of the orders stored by the server. The controller function makes an Ajax request to that URL and assigns the data objects to the `orders` property on the scope or, if the request is unsuccessful, assigns the error object.

The rest of the controller is straightforward. The `selectOrder` behavior is called to set a `selectedOrder` property, which I will use to zoom in on the details of an order. The `calcTotal` behavior works out the total value of the products in an order.

To take advantage of the `ordersCtrl` controller, I have removed the placeholder content from the `adminOrders.html` file and replaced it with the markup shown in Listing 8-19.

Listing 8-19. The Contents of the adminOrders.html File

```
<div ng-controller="ordersCtrl">

    <table class="table table-striped table-bordered">
        <tr><th>Name</th><th>City</th><th>Value</th><th></th></tr>
        <tr ng-repeat="order in orders">
            <td>{{order.name}}</td>
            <td>{{order.city}}</td>
            <td>{{calcTotal(order) | currency}}</td>
            <td>
                <button ng-click="selectOrder(order)" class="btn btn-xs btn-primary">
                    Details
                </button>
            </td>
        </tr>
    </table>

    <div ng-show="selectedOrder">
        <h3>Order Details</h3>

        <table class="table table-striped table-bordered">
            <tr><th>Name</th><th>Count</th><th>Price</th></tr>
            <tr ng-repeat="item in selectedOrder.products">
                <td>{{item.name}}</td>
                <td>{{item.count}}</td>
                <td>{{item.price| currency}} </td>
            </tr>
        </table>
    </div>
</div>
```

The view consists of two `table` elements. The first table shows a summary of the orders, along with a `button` element that invokes the `selectOrder` behavior to focus on the order. The second table is visible only once an order has been selected and displays details of the products that have been ordered. You can see the result in Figure 8-12.

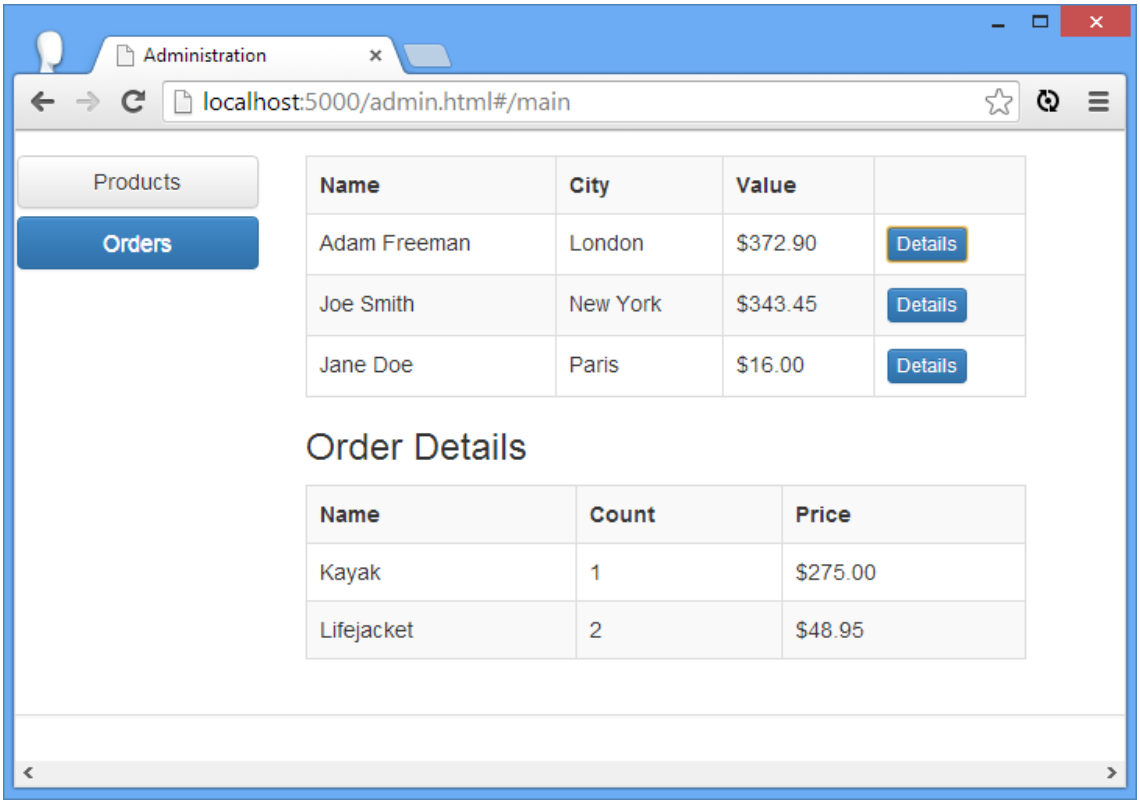

Figure 8-12. Viewing the SportsStore orders

Implementing the Products Feature

For the products feature, I am going to perform a full range of operations on the data so that the administrator not only can see the products but create new ones and edit and delete existing ones. Parse.com provides a RESTful API whose actions you configured earlier in this chapter. I get into the details of RESTful APIs properly in Chapter 21, but the short version is that the data object you want is specified using the URL, and the operation you want to perform is specified by the HTTP method of the request sent to the server. So, for example, if I want to delete the object whose `objectId` attribute is `100`, I would sent a request to the server using the DELETE HTTP method and the URL `/1/classes/Products/100`.

You can use the `$http` service to work with a RESTful API, but doing so means you have to expose the complete set of URLs that are used to perform operations throughout the application. You can do this by defining a service that performs the operations for you, but a more elegant alternative is to use the `$resource` service, defined in the optional `ngResource` module, which also has a nice way of dealing with defining the URLs that are used to send requests to the server.

Defining the RESTful Controller

I am going to start by defining the controller that will provide access to the RESTful Parse.com API via the AngularJS `$resource` service. I created a new file called `adminProductController.js` in the `controllers` folder and used it to define the controller shown in Listing 8-20.

Listing 8-20. The Contents of the adminProductController.js File

```
angular.module("sportsStoreAdmin")
.constant("productUrl", "https://api.parse.com/1/classes/Products/")
.run(function ($http) {
    $http.defaults.headers.common["X-Parse-Application-Id"]
        = "McOi2Q6FZNovOr4bQD6QRFwzgw4iRHOy2JcDYuLq";
    $http.defaults.headers.common["X-Parse-REST-API-Key"]
        = "h4dnGScEtCWMMjaWmCJYNBZrDCrnROZmsKvEwXLD";
})
.controller("productCtrl", function ($scope, $http, $resource, productUrl) {

    $scope.$on("sessionToken", function (sessionToken) {
        $http.defaults.headers.common["X-Parse-Session-Token"] = sessionToken;
    });

    function getData(data, headers) {
        return JSON.parse(data).results;
    }

    $scope.productsResource = $resource(productUrl + ":id", { id: "@objectId" }, {
        query: { method: "GET", isArray: true, transformResponse: getData },
        update: { method: "PUT"}
    });

    $scope.listProducts = function () {
        $scope.products = $scope.productsResource.query();
    }

    $scope.deleteProduct = function (product) {
        product.$delete().then(function () {
            $scope.products.splice($scope.products.indexOf(product), 1);
        });
    }

    $scope.createProduct = function (product) {
        new $scope.productsResource(product).$save().then(function (response) {
            response.$get().then(function (newProduct) {
                $scope.products.push(newProduct);
                $scope.editedProduct = null;
            })
        });
    }

    $scope.updateProduct = function (product) {
        var pCopy = {};
        angular.copy(product, pCopy)
        pCopy.$update().then(function () {
            $scope.editedProduct = null;
        })
```

```
    }

    $scope.startEdit = function (product) {
        $scope.editedProduct = product;
    }

    $scope.cancelEdit = function () {
        $scope.editedProduct = null;
    }

    $scope.listProducts();
});
```

I am not going to into the code for this listing because I cover the topic fully in Chapter 21. But there some important themes that are worth explaining now, so I'll cover just the highlights.

The most important part of this example, however, is the statement that creates the *access object* that provides access to the RESTful API:

```
...
$scope.productsResource = $resource(productUrl + ":id", { id: "@objectId" }, {
    query: { method: "GET", isArray: true, transformResponse: getData },
    update: { method: "PUT"}
 });
...
```

The first argument passed into the `$resource` call defines the URL format that will be used to make queries. The `:id` part, which corresponds to the map object that is the second argument, tells AngularJS that if the data object it is working with has an `objectId` property, then it should be appended to the URL used for the Ajax request.

The URLs and HTTP methods that are used to access the RESTful API are inferred from these two arguments, which means I don't have to make individual Ajax calls using the `$http` service.

The access object that is the result from using the `$resource` service has `query`, `get`, `delete`, `remove`, and `save` methods that are used to obtain and operate on the data from the server (methods are also defined on individual data objects, as I explain in Chapter 21). Calling these methods triggers the Ajax request that performs the required operation.

The final argument that I pass to the `$resource` method is a configuration object that lets me define new operations on the server and change the existing ones. I have defined an `update` operation that uses an HTTP PUT request to update a modified object and adjusted the `query` operation to specify a function that extracts the data that AngularJS is expecting from the response send by Parse.com.

Tip The methods defined by the access object don't quite correspond to the API defined by Parse.com, which is why I have had to add a new operation and modify an existing one. In Chapter 21, I show you how you can change the `$resource` configuration to fully map onto any RESTful API.

Most of the code in the controller presents these methods to the view in a useful way that works around a wrinkle in the `$resource` implementation. The collection of data objects returned

by the `query` method isn't automatically updated when objects are created or deleted, so I have to include code to take care of keeping the local collection in sync with the remote changes.

Tip The access object doesn't automatically load the data from the server, which is why I call the `query` method directly at the end of the controller function.

Defining the View

To take advantage of the functionality defined by the controller, I have replaced the placeholder content in the `adminProducts.html` view with the markup shown in Listing 8-21.

Listing 8-21. The Contents of the adminProducts.html File

```
<style>
    #productTable { width: auto; }
    #productTable td { max-width: 150px; text-overflow: ellipsis;
                       overflow: hidden; white-space: nowrap; }
    #productTable td input { max-width: 125px; }
</style>

<div ng-controller="productCtrl">
    <table id="productTable" class="table table-striped table-bordered">
        <tr>
            <th>Name</th>
            <th>Description</th>
            <th>Category</th>
            <th>Price</th>
            <th></th>
        </tr>
        <tr ng-repeat="item in products | orderBy: 'category'"
                ng-hide="item.objectId == editedProduct.objectId">
            <td>{{item.name}}</td>
            <td class="description">{{item.description}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price | currency}}</td>
            <td>
                <button ng-click="startEdit(item)" class="btn btn-xs btn-primary">
                    Edit
                </button>
                <button ng-click="deleteProduct(item)" class="btn btn-xs btn-primary">
                    Delete
                </button>
            </td>
        </tr>
        <tr ng-class="{danger: editedProduct}">
            <td><input ng-model="editedProduct.name" required /></td>
            <td><input ng-model="editedProduct.description" required /></td>
            <td><input ng-model="editedProduct.category" required /></td>
            <td><input type="number" ng-model="editedProduct.price" required /></td>
            <td>
                <button ng-hide="editedProduct.objectId"
```

```
                    ng-click="createProduct(editedProduct)"
                    class="btn btn-xs btn-primary">
                Create
            </button>
            <button ng-show="editedProduct.objectId"
                    ng-click="updateProduct(editedProduct)"
                    class="btn btn-xs btn-primary">
                Save
            </button>
            <button ng-show="editedProduct"
                    ng-click="cancelEdit()" class="btn btn-xs btn-primary">
                Cancel
            </button>
        </td>
    </tr>
</table>
</div>
```

There are no new techniques in this view, but it shows how AngularJS directives can be used to manage a stateful editor view. The elements in the view use the controller behaviors to manipulate the collection of product objects, allowing the user to create new products and edit or delete existing products.

Adding the References to the HTML File

All that remains is to add `script` elements to the `admin.html` file to import the new module and the new controller and to update the main application module so that it declares a dependency on `ngResource`, as shown in Listing 8-22.

Listing 8-22. Adding the References to the admin.html File

```
<!DOCTYPE html>
<html ng-app="sportsStoreAdmin">
<head>
    <title>Administration</title>
    <script src="angular.js"></script>
    <script src="ngmodules/angular-route.js"></script>
    <script src="ngmodules/angular-resource.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("sportsStoreAdmin", ["ngRoute", "ngResource"])
            .config(function ($routeProvider) {

                $routeProvider.when("/login", {
                    templateUrl: "/views/adminLogin.html"
                });

                $routeProvider.when("/main", {
                    templateUrl: "/views/adminMain.html"
                });

                $routeProvider.otherwise({
                    redirectTo: "/login"
```

```
            });
        });
    </script>
    <script src="controllers/adminControllers.js"></script>
    <script src="controllers/adminProductController.js"></script>
</head>
<body>
    <ng-view />
</body>
</html>
```

You can see the effect in Figure 8-13. The user creates a new product by filling in the `input` elements and clicking the Create button, modifies a product by clicking one of the Edit buttons, and removes a product using one of the Delete buttons.

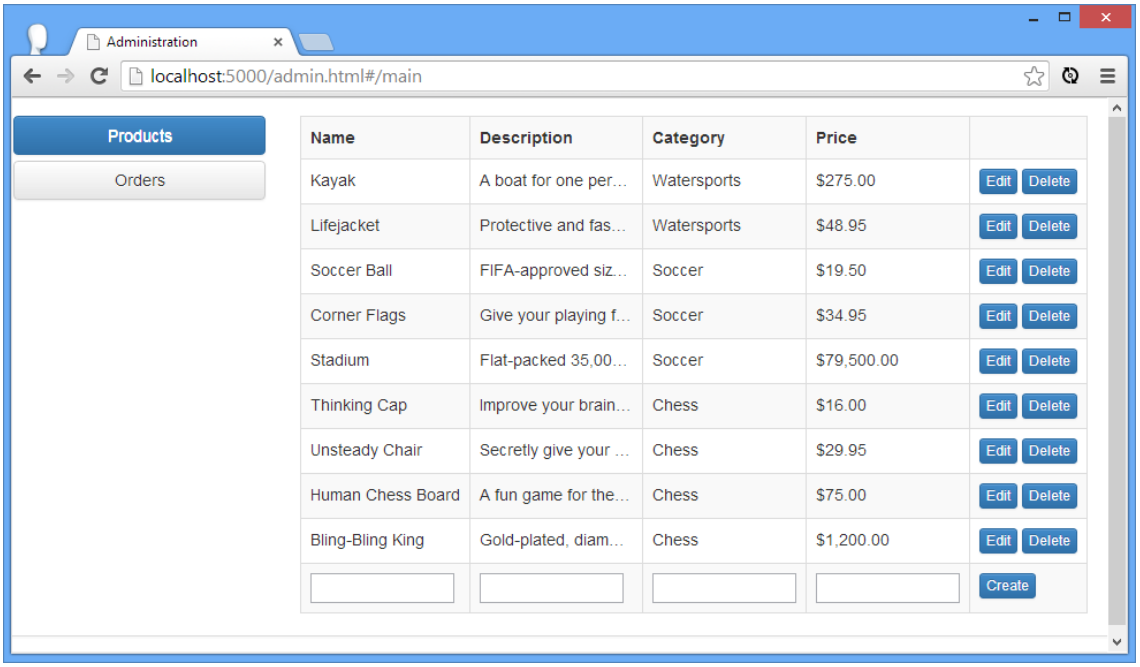

Figure 8-13. Editing products

Summary

In this chapter, I finished the main SportsStore application and built the SportsStore administration tool. I showed you how to perform form validation, showed how to make Ajax POST requests with the `$http` service, and outlined some improvements that could be made using some of the advanced techniques I describe in later chapters. For the administration application, I showed you how to perform authentication (and configure Ajax requests so that they work with security cookies) and how to use the `$resource` service to consume a RESTful API. You will see the features and themes that I used in the SportsStore application throughout

the rest of this book. In Part 2, I dig into AngularJS in detail, starting with an overview of the different AngularJS components.

CHAPTER 21

Services for REST

In this chapter, I show you how AngularJS supports working with RESTful web services. *Representational State Transfer* (REST) is a style of API that operates over HTTP requests, which I introduced in Chapter 3. The requested URL identifies the data to be operated on, and the HTTP method identifies the operation that is to be performed.

REST is a style of API rather than a formal specification, and there is a lot of debate and disagreement about what is and isn't RESTful, a term used to indicate an API that follows the REST style. AngularJS is pretty flexible about how RESTful web services are consumed, and I show you how you can tailor AngularJS to work with specific REST implementations.

Don't worry if you are not familiar with REST or if you have not worked with a RESTful web service before. I start by building a simple REST service and then provide plenty of examples to show you how to use it. Table 21-1 summarizes this chapter.

Table 21-1. *Chapter Summary*

Problem	Solution	Listing
Consume a RESTful API through explicit Ajax requests.	Use the `$http` service to request the data from the server and perform operations on it.	1–8
Consume a RESTful API without exposing the Ajax requests.	Use the `$resource` service.	9–14
Tailor the Ajax requests used by the `$resource` service.	Define custom actions or redefine the default ones.	15–16
Create components that can work with RESTful data.	Ensure that you can optionally enable support for working with the `$resource` service and remember to allow the actions that must be used to be configured when the component is applied.	17–18

Why and When to Use the REST Services

You should use the services that I describe in this chapter when you are performing data operations on a RESTful API. You may initially prefer to use the `$http` service to make Ajax requests, especially if you are coming from a jQuery background. To that end, I describe the use

of `$http` at the start of the chapter, before explaining its limitations when used with REST and the advantages of using the `$resource` service as an alternative.

Preparing the Example Project

I need a back-end service to demonstrate the different ways in which AngularJS can be used to consume a RESTful web service, so I will be using Parse.com once again and creating a similar – but simpler – web service that I can use with AngularJS.

Creating the RESTful Service

Log into the Parse.com website using the account you created in Chapter 6 and click on the Create New App button and enter `SportsStoreLite` into the App Name field as shown in Figure 21-1.

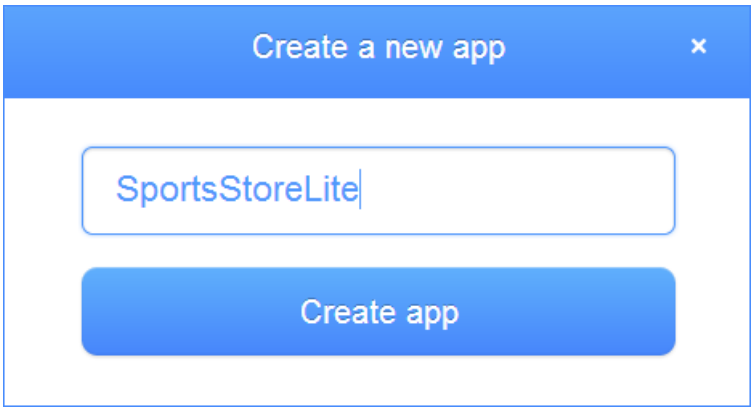

Figure 21-1. Creating a New Parse.com Application

Click the Create App button and make a note of the Application, JavaScript and REST API keys. Each application has its own keys, but Table 21-2 shows the keys that I received when created the example.

Table 21-2. *The Parse.com Keys Required for the AngularJS SportsStoreLite Application*

Key	Value
Application ID	`7uRKj9WGCnHTZ136SAbh7N3TBI3NyWrpgWTgomjp`
REST API Key	`aeZgyAPTIKbs6YAsCNyySuORW53xeKrPC6DUUqSw`
JavaScript Key	`wT4w53fSYsjC2yx1BlJo2VZfUmE38ijA7pAerTQf`

Click the Data Browser button to display the data grid for the application, which is presently empty.

Creating the Data Structure

Click the New Class button, ensure that `Custom` is selected and enter `Products` into the text field. Click the Create Class button to create the `Products` collection and then use the +Col button to define the columns shown in Table 21-3.

Table 21-3. *The Columns Required for the Products Class*

Name	Type
name	String
category	String
price	Number

Adding the Initial Data

I am going to populate the back end with some initial data to make creating the example simple. Use the +Row button to create data rows for the values shown in Table 21-4. Ignore the built-in rows that Parse.com adds to all objects – values will be assigned to them automatically.

Tip You can drag columns in the data grid to bring the custom rows together, which makes entering data easier.

Table 21-4. *The Initial Data Items*

Name	Category	Price
Apples	Fruit	1.20
Bananas	Fruit	2.42
Pears	Fruit	2.02
Tuna	Fish	20.45
Salmon	Fish	17.93
Trout	Fish	12.93

When you have added the data, the data browser grid should look like the one shown in Figure 21-2. (I have omitted most of the built-in properties from the figure).

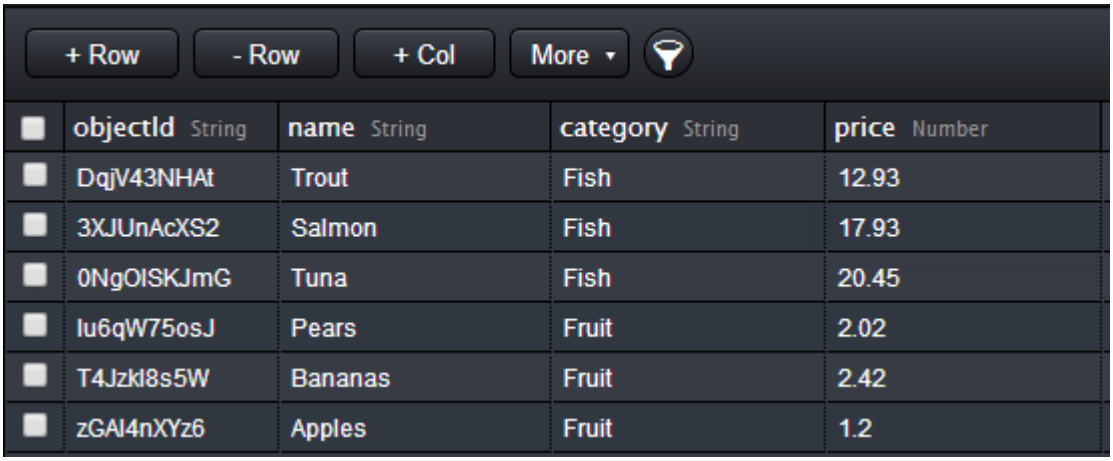

Figure 21-2. Adding the data

Testing the API

My goal in this chapter is to show you the different facilities that AngularJS provides to combine these URLs and HTTP methods to drive the data from an application. In Table 21-5, I have listed the operations that can be performed on the Parse.com data and the combination of HTTP method, URL and data that each requires.

Tip It is always worth checking the API that your RESTful service provides because there isn't complete consistency about the way that HTTP methods are combined with URLs to manipulate data. As an example, some services support using the `PATCH` method to update individual properties for an object, whereas others, including Parse.com, use the `PUT` method.

Table 21-5. *The HTTP Methods and URLs That the RESTful Service Supports*

Task	Method	URL	Accepts	Returns
List products	`GET`	`/1/classes/Products`	Nothing	An array of objects
Create an object	`POST`	`/1/classes/Products`	A single object	The saved object
Get an object	`GET`	`/1/classes/Products/<id>`	Nothing	A single object
Update an object	`PUT`	`/1/classes/Products/<id>`	A single object	The saved object
Delete an object	`DELETE`	`/1/classes/Products/<id>`	A single object	Nothing

As I explained in Chapter 6, Parse.com identifies the application that a request relates to using HTTP request headers whose values are the keys in Table 2. It is possible to test the list operation in the browser by using the application and JavaScript keys in the URL, in the following format:

```
https://<appid>:javascript-key=<jskey>@api.parse.com/1/classes/Products
```

where `<appid>` is the value of the Application ID from Table 6-2 and `<jskey>` is the value of the JavaScript key (although you will have different values from the ones I have listed since each application is given unique keys).

Caution The class name in the Parse.com URL is case-sensitive. Ensure that your URL includes `Products` and not `products`.

Using my key values I would request the following URL to test my Parse.com backend service:

```
https://7uRKj9WGCnHTZ136SAbh7N3TBI3NyWrpgWTgomjp:javascript-
key=wT4w53fSYsjC2yx1BlJo2VZfUmE38ijA7pAerTQf@api.parse.com/1/classes/Products/
```

When this URL is requested, the Parse.com server returns a JSON string that contains the details entered from Table 21-4. If you are using Google Chrome, then the JSON will be displayed in the browser window, but other browsers, including Internet Explorer, will ask you to save the JSON data to a file. The JSON from Parse.com is similar to the JSON I manually created in Chapter 20, but with two differences: Since the data is being stored in a database, each product object is assigned a unique key on a property called `objectId` and the collection of data objects is defined in an object with a `results` property. Here is the first part of the JSON sent by Parse.com with both the `objectId` and `results` property highlighted:

```
{"results":[
    {"category":"Fruit",
     "name":"Apples",
     "price":1.2,
     "createdAt":"2014-04-27T07:22:01.635Z",
     "updatedAt":"2014-04-27T07:22:12.072Z",
     "objectId":"zGAI4nXYz6"},

     ...other data items omitted for brevity...
]}
```

The objectId value zGAI4nXYz6 uniquely identifies the product object whose name property is set to Apples. To delete this object via REST, I would use the HTTP DELETE method to invoke the following URL:

```
https://api.parse.com/1/classes/Products/b57776c8bd96ba29
```

Creating the AngularJS Application

Now that the RESTful API is set up and populated with data, I am going to create a skeletal AngularJS application. This application will display the content and present the user with the means to add, modify, and delete product objects.

I started by clearing the contents of the angularjs directory and reinstalling the AngularJS and Bootstrap files, as described in Chapter 1. I then created a new HTML file called products.html, the contents of which you can see in Listing 21-1.

Listing 21-1. The Contents of the products.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Products</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="products.js"></script>
</head>
<body ng-controller="defaultCtrl">
    <div class="panel panel-primary">
        <h3 class="panel-heading">Products</h3>
        <ng-include src="'tableView.html'" ng-show="displayMode == 'list'"></ng-include>
        <ng-include src="'editorView.html'" ng-show="displayMode == 'edit'"></ng-include>
    </div>
</body>
</html>
```

I am going to break this example into a series of smaller files, much as you would do in a real project. The products.html file contains the script element for AngularJS and the link elements for Bootstrap. The main content for this application is contained in two view files, tableView.html and editorView.html, which I will create shortly. These are imported into the products.html file using the ng-include directive, and the visibility of the elements is controlled using the ng-show directive tied to a scope variable called displayMode.

The products.html file also contains a script element for a file called products.js, which I have used to define the behaviors that the application will need. I have started by using dummy local data, which I will replace with data obtained via REST later in the chapter. Listing 21-2 shows the contents of the products.js file.

Listing 21-2. The Contents of the products.js File

```
angular.module("exampleApp", [])
```

```
.controller("defaultCtrl", function ($scope) {

    $scope.displayMode = "list";
    $scope.currentProduct = null;

    $scope.listProducts = function () {
        $scope.products = [
            { objectId: 0, name: "Dummy1", category: "Test", price: 1.25 },
            { objectId: 1, name: "Dummy2", category: "Test", price: 2.45 },
            { objectId: 2, name: "Dummy3", category: "Test", price: 4.25 }];
    }

    $scope.deleteProduct = function (product) {
        $scope.products.splice($scope.products.indexOf(product), 1);
    }

    $scope.createProduct = function (product) {
        product.objectId = $scope.products.length;
        $scope.products.push(product);
        $scope.displayMode = "list";
    }

    $scope.updateProduct = function (product) {
        for (var i = 0; i < $scope.products.length; i++) {
            if ($scope.products[i].objectId == product.objectId) {
                $scope.products[i] = product;
                break;
            }
        }
        $scope.displayMode = "list";
    }

    $scope.editOrCreateProduct = function (product) {
        $scope.currentProduct = product || {};
        $scope.displayMode = "edit";
    }

    $scope.saveEdit = function (product) {
        if (angular.isDefined(product.objectId)) {
            $scope.updateProduct(product);
        } else {
            $scope.createProduct(product);
        }
    }

    $scope.cancelEdit = function () {
        $scope.currentProduct = {};
        $scope.displayMode = "list";
    }

    $scope.listProducts();
});
```

The controller in the listing defines all the functionality I need to operate on the product data. The behaviors I have defined fall into two categories. The first category consists of behaviors that manipulate the data in the scope: the `listProducts`, `deleteProduct`, `createProduct`,

and updateProduct functions. These behaviors correspond to the REST operations I described in Table 21-5, and most of this chapter is spent showing you different ways to implement those methods. For the moment, the application uses some dummy test data, just so I can separate showing you how the application works from showing you how to consume restful services.

The other behaviors, editOrCreateProduct, saveEdit, and cancelEdit, all support the user interface and are invoked in response to user interaction. In Listing 21-1, you will see that I used the ng-include directive to import two HTML views. The first of these is called tableView.html, and I use it to display the data and provide buttons that will allow the user to reload the data and create, delete, and edit a product. Listing 21-3 shows the contents of the tableView.html file.

Listing 21-3. The Contents of the tableView.html File

```
<div class="panel-body">
    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th>
                <th>Category</th>
                <th class="text-right">Price</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="item in products">
                <td>{{item.name}}</td>
                <td>{{item.category}}</td>
                <td class="text-right">{{item.price | currency}}</td>
                <td class="text-center">
                    <button class="btn btn-xs btn-primary"
                            ng-click="deleteProduct(item)">
                        Delete
                    </button>
                    <button class="btn btn-xs btn-primary"
                            ng-click="editOrCreateProduct(item)">
                        Edit
                    </button>
                </td>
            </tr>
        </tbody>
    </table>
    <div>
        <button class="btn btn-primary" ng-click="listProducts()">Refresh</button>
        <button class="btn btn-primary" ng-click="editOrCreateProduct()">New</button>
    </div>
</div>
```

This view uses AngularJS features that I have described in earlier chapters. I use the ng-repeat directive to generate rows in a table for each product object, and I use the currency filter to format the price property on the product objects. Finally, I use the ng-click directive to respond when the user clicks a button, calling the behaviors defined in the controller defined in the products.js file.

The other view file is called `editorView.html`, and I use it to allow the user to create new product objects or edit existing ones. You can see the contents of the `editorView.html` file in Listing 21-4.

Listing 21-4. The Contents of the editorView.html File

```
<div class="panel-body">
    <div class="form-group">
        <label>Name:</label>
        <input class="form-control" ng-model="currentProduct.name" />
    </div>
    <div class="form-group">
        <label>Category:</label>
        <input class="form-control" ng-model="currentProduct.category" />
    </div>
    <div class="form-group">
        <label>Price:</label>
        <input type="number" class="form-control" ng-model="currentProduct.price" />
    </div>
    <button class="btn btn-primary" ng-click="saveEdit(currentProduct)">Save</button>
    <button class="btn btn-primary" ng-click="cancelEdit()">Cancel</button>
</div>
```

This view uses the `ng-model` directive to create two-way bindings with the product being edited or created, and it uses the `ng-click` directive to respond to the user clicking the Save or Cancel button.

Testing the Application

To test the AngularJS application, simply load the `products.html` file into the browser. All of the other files will be imported, and you will see the list of dummy data, as illustrated in Figure 21-3.

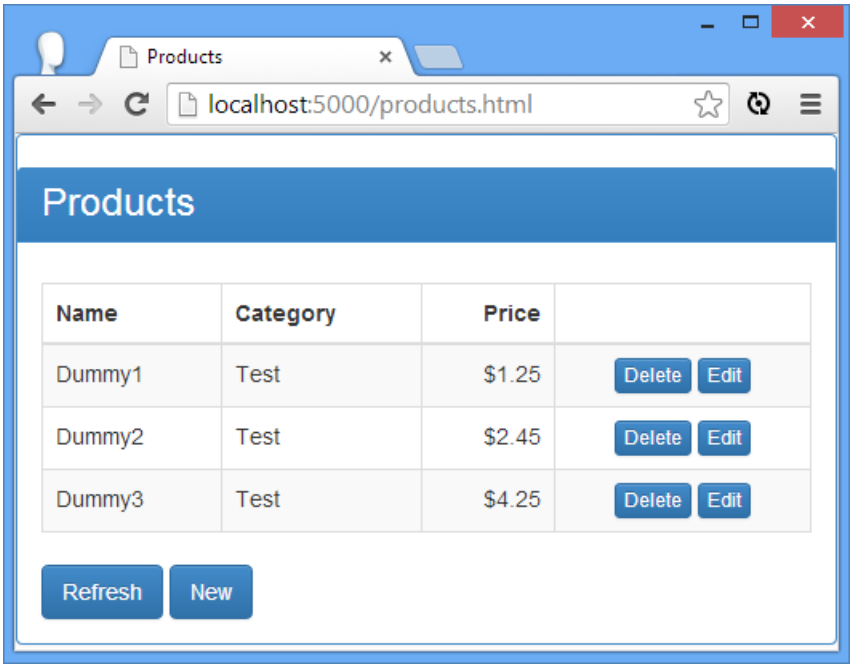

Figure 21-3. Displaying dummy data

If you click the Delete button, the `deleteProduct` behavior will be invoked, and the product in the corresponding row will be removed from the data array. If you click the Refresh button, the `listProducts` behavior will be invoked, and the data will be reset because this is where the dummy data is defined; the data won't be reset when I start making Ajax requests.

Clicking the Edit or New button will invoke the `editOrCreateProduct` behavior, which causes the contents of the `editorView.html` file to be displayed, as shown in Figure 21-4.

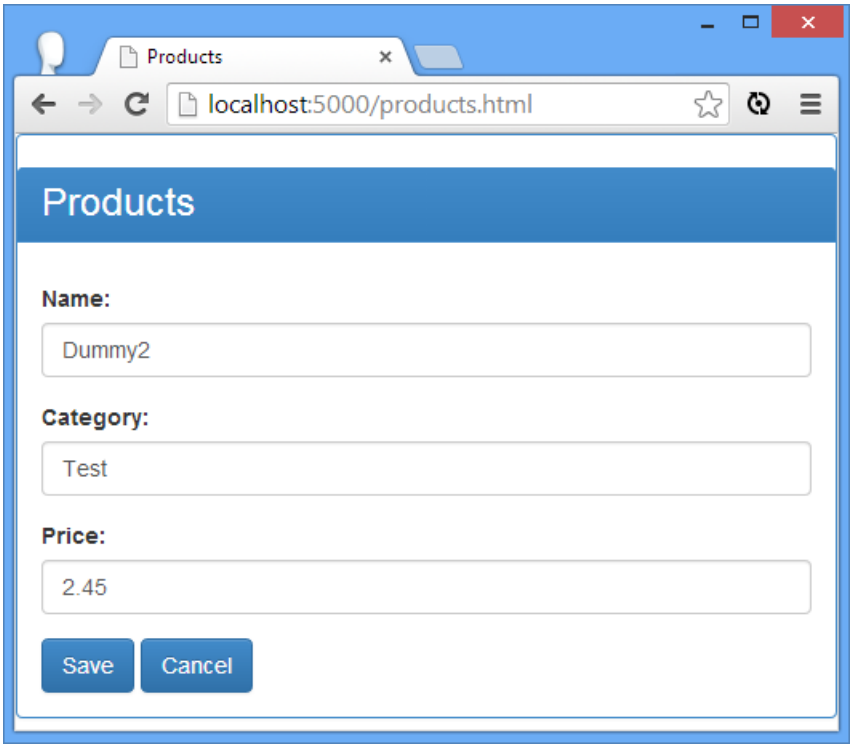

Figure 21-4. Editing or creating a product

If you click the Save button, the changes made to an existing item will be saved or a new product will be created. I rely on the fact that data objects that are being edited will have an `objectId` attribute. The Cancel button returns to the list view without saving any changes, which I handle by using the `angular.copy` method to create a copy of the `product` object so that I can discard it when needed.

Using the $http Service

The first service that I am going to use to complete the implementation of the example application is `$http`, which I described in Chapter 20. RESTful services are consumed using standard asynchronous HTTP requests, and the `$http` service provides all of the features that are required to bring the data into the application and write changes to the server. In the sections that follow, I'll rewrite each of the data manipulation behaviors to use the `$http` service.

Listing the Product Data

None of the changes that I have to make to use Ajax is especially complex, and in Listing 21-5, you can see how I have changed the definition of the controller factory function to declare its dependencies.

I don't want to embed the URL for the RESTful service throughout the application, so I have defined a constant called `baseUrl` for the root URL that provides access to the data.

I then declare a dependency on `baseUrl` (which is possible because, as I explained in Chapter 18, constants are just simple services). I have also used the `run` method to set the headers that provide Parse.com with the key values it required and to define an Ajax interceptor that extracts the data array returned by Parse.com from the `results` property.

Caution Remember to change the keys to the ones you received when creating the application at the start of the chapter.

Listing 21-5. Declaring Dependencies and Listing Data in the products.js File

```
angular.module("exampleApp", [])
.constant("baseUrl", "https://api.parse.com/1/classes/Products/")
.config(function ($httpProvider) {
    $httpProvider.defaults.headers.common["X-Parse-Application-Id"]
        = "7uRKj9WGCnHTZ136SAbh7N3TBI3NyWrpgWTgomjp";
    $httpProvider.defaults.headers.common["X-Parse-REST-API-Key"]
        = "aeZgyAPTIKbs6YAsCNyySuORW53xeKrPC6DUUqSw";

    $httpProvider.interceptors.push(function () {
        return {
            response: function (response) {
                if (response.headers("content-type").indexOf("application/json") != -1) {
                    if (response.hasOwnProperty("data")
                            && response.data.hasOwnProperty("results")) {
                        response.data = response.data.results;
                    }
                }
                return response;
            }
        }
    })
})
.controller("defaultCtrl", function ($scope, $http, baseUrl) {

    $scope.displayMode = "list";
    $scope.currentProduct = null;

    $scope.listProducts = function () {
        $http.get(baseUrl).success(function (data) {
            $scope.products = data;
        });
    }

    $scope.deleteProduct = function (product) {
        $scope.products.splice($scope.products.indexOf(product), 1);
    }

    $scope.createProduct = function (product) {
```

```
        product.objectId = $scope.products.length;
        $scope.products.push(product);
        $scope.displayMode = "list";
    }

    $scope.updateProduct = function (product) {
        // do nothing - no action required for local data
        $scope.displayMode = "list";
    }

    $scope.editOrCreateProduct = function (product) {
        $scope.currentProduct = product || {};
        $scope.displayMode = "edit";
    }

    $scope.saveEdit = function (product) {
        if (angular.isDefined(product.objectId)) {
            $scope.updateProduct(product);
        } else {
            $scope.createProduct(product);
        }
    }

    $scope.cancelEdit = function () {
        $scope.currentProduct = {};
        $scope.displayMode = "list";
    }

    $scope.listProducts();
});
```

The implementation of the `listProduct` method relies on the `$http.get` convenience method that I described in Chapter 20. I make a call to the base URL, which, as Table 21-5 noted, obtains the array of `product` objects from the server. I use the `success` method to receive the data that the server sends and assign it to the `products` property in the controller scope.

The last statement in the controller's factory function calls the `listProduct` behavior to ensure that the application starts with some data. You can see the effect by loading `products.html` into the browser and using the F12 developer tools to look at the network requests that are made. You will see a GET request being made to the base URL, and the data will be displayed in the `table` element, as shown in Figure 21-5.

Tip You may notice a small delay between the contents of the `tableView.html` file being displayed and the `table` element being populated. This is the time taken for the server to process the Ajax request and send the response, and it can be quite pronounced when the network or the service is busy. In Chapter 22, I show you how you can use the URL routing feature to prevent the view from being shown until the data has arrived.

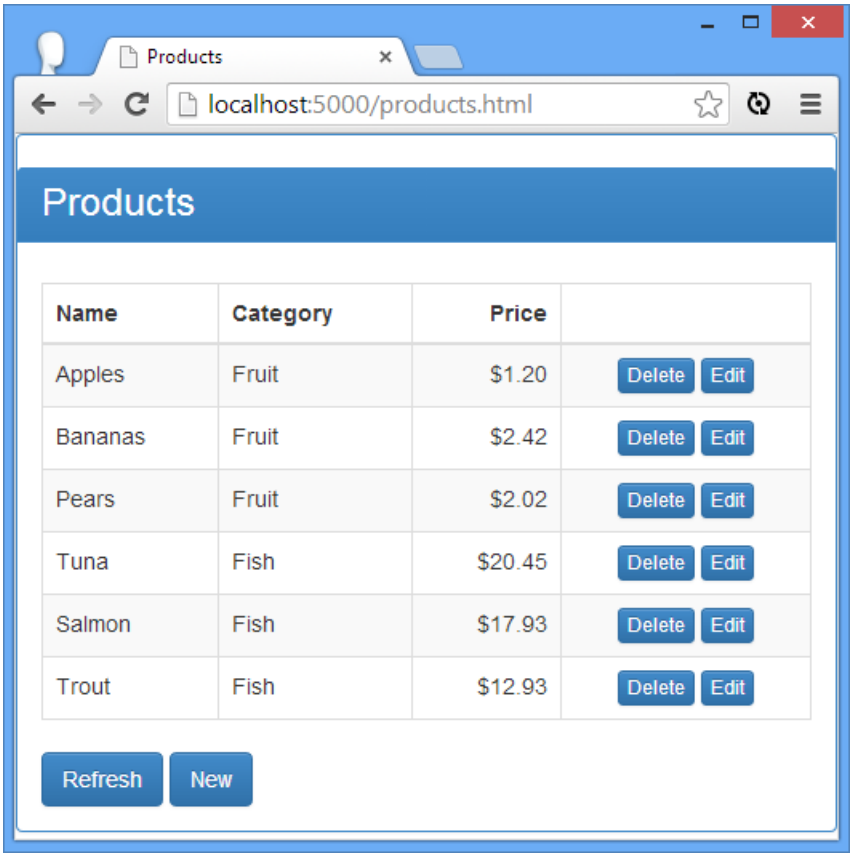

Figure 21-5. Listing the data from the server using Ajax

Deleting Products

The next behavior I am going to re-implement is `deleteProduct`, which you can see in Listing 21-6.

Listing 21-6. Adding Ajax Requests to the deleteProduct Function in the products.js File

```
...
$scope.deleteProduct = function (product) {
    $http({
        method: "DELETE",
        url: baseUrl + product.objectId
    }).success(function () {
        $scope.products.splice($scope.products.indexOf(product), 1);
    });
}
...
```

There is no `$http` convenience method for the HTTP `DELETE` method, so I have to use the alternative technique of treating the `$http` service object as a function and pass in a

configuration object. I described the properties that can be set on a configuration object in Chapter 20, but I need only the `method` and `url` properties for this example.

I set the URL to be the base URL plus the `objectId` of the product I want deleted following the URL pattern I listed in Table 21-5. The `$http` service object returns a promise, and I use the `success` method to delete the corresponding `product` from the local array so that the server data and the local copy of it remain in sync.

The effect of this change is that clicking a Delete button removes the corresponding `product` from the server and the client. You can see the change both in the Parse.com Data Browser and, of course, in the browser that is running the example AngularJS application.

Creating Products

Adding support for creating new `product` objects requires the use of the HTTP `POST` method, for which there is an `$http` convenience method. You can see the changes I made to the `createProduct` behavior in Listing 21-7.

Listing 21-7. Creating Products in the products.js File

```
...
$scope.createProduct = function (product) {
    $http.post(baseUrl, product).success(function (response) {
        product.objectId = response.objectId;
        $scope.products.push(product);
        $scope.displayMode = "list";
    });
}
...
```

The RESTful service responds to my create request by returning an object that has an `objectId` property that identifies the newly created data object at the server. I use the value from the response to set the `objectId` of the `product` parameter object, which I then add to the `products` array. Once I have added the new object to the array, I set the `displayMode` variable so that the application displays the list view.

Updating Products

The last behavior I have to revise is `updateProduct`, which you can see in Listing 21-8.

Listing 21-8. Using Ajax in the updateProduct Behavior Defined in the product.json File

```
...
$scope.updateProduct = function (product) {
    $http({
        url: baseUrl + product.objectId,
        method: "PUT",
        data: product
    }).success(function () {
        for (var i = 0; i < $scope.products.length; i++) {
            if ($scope.products[i].objectId == product.objectId) {
                $scope.products[i] = product;
```

```
                break;
            }
        }
        $scope.displayMode = "list";
    });
}
...
```

Updating an existing `product` object requires the HTTP `PUT` method for which there is no `$http` convenience method, meaning that I have to invoke the `$http` service object as a function and pass in a configuration object with the method and URL. The response from the server just a confirmation of the update, so I use the `product` parameter passed to the method to update the `products` array to reflect the changes. Once I have added the modified object to the array, I set the `displayMode` variable so that the application displays the list view.

Testing the Ajax Implementation

You can see from the previous sections that implementing the Ajax calls to integrate the RESTful service into the application is a relatively simple task. I have skipped over some details that would be required in a real application, such as form validation and handling errors, but you get the idea: With just a little care and thought, it is easy to use the `$http` service to consume a RESTful service.

Hiding the Ajax Requests

Using the `$http` service to consume a RESTful API is easy, and it provides a nice demonstration of how different AngularJS features can be combined to create applications. In terms of features, it works just fine, but there are serious problems when it comes to the design of the application that it produces.

The problem is that the local data and the behaviors that manipulate the data on the server are separate and care has to be taken to make sure that they stay synchronized. This runs counter to the way that AngularJS usually work, where data is propagated throughout the application via scopes and can be updated freely. To demonstrate the problem, I have added a new file to the `angularjs` folder called `increment.js`, which contains the module shown in Listing 21-9.

Listing 21-9. The Contents of the increment.js File

```
angular.module("increment", [])
    .directive("increment", function () {
        return {
            restrict: "E",
            scope: {
                value: "=value"
            },
            link: function (scope, element, attrs) {
                var button = angular.element("<button>").text("+");
                button.addClass("btn btn-primary btn-xs");
                element.append(button);
                button.on("click", function () {
```

```
                scope.$apply(function () {
                    scope.value++;
                })
            })
        },
    }
});
```

The module in this file, called `increment`, contains a directive, also called `increment`, that updates a value when the button is clicked. The directive is applied as an element and uses a two-way binding on an isolated scope to get its data value (a process that I described in Chapter 16). To use the module, I had to add a `script` element to the `products.html` file, as shown in Listing 21-10.

Listing 21-10. Adding a script Element to the products.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Products</title>
    <script src="angular.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="products.js"></script>
    <script src="increment.js"></script>
</head>
<body ng-controller="defaultCtrl">
    <div class="panel panel-primary">
        <h3 class="panel-heading">Products</h3>
        <ng-include src="'tableView.html'" ng-show="displayMode == 'list'"></ng-include>
        <ng-include src="'editorView.html'" ng-show="displayMode == 'edit'"></ng-include>
    </div>
</body>
</html>
```

I also had to add a dependency for the module in the `products.js` file, as shown in Listing 21-11.

Listing 21-11. Adding a Module Dependency in the products.js File

```
angular.module("exampleApp", ["increment"])
.constant("baseUrl", "https://api.parse.com/1/classes/Products/")
.config(function ($httpProvider) {
...
```

And, finally, I had to apply the directive to the `tableView.html` file so that each row in the table has an increment button, as shown in Listing 21-12.

Listing 21-12. Applying the increment Directive to the tableView.html File

```
...
<tr ng-repeat="item in products">
    <td>{{item.name}}</td>
    <td>{{item.category}}</td>
    <td class="text-right">{{item.price | currency}}</td>
```

```
        <td class="text-center">
            <button class="btn btn-xs btn-primary"
                    ng-click="deleteProduct(item)">
                Delete
            </button>
            <button class="btn btn-xs btn-primary"
                    ng-click="editOrCreateProduct(item)">
                Edit
            </button>
            <increment value="item.price" />
        </td>
    </tr>
    ...
```

The effect is shown in Figure 21-6. Clicking the + button increments the `price` property of the corresponding `product` object by `1`.

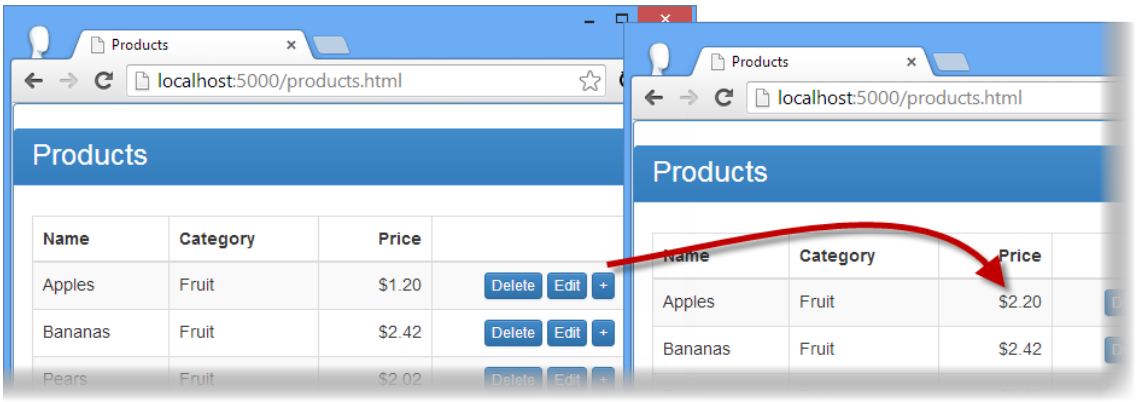

Figure 21-6. Incrementing prices

The problem can be seen by clicking the Refresh button, which replaces the local product data with fresh data from the server. The `increment` directive didn't perform the required Ajax update when it incremented the `price` property, so the local data fell out of sync with the server data.

This may seem like a contrived example, but it arises frequently when using directives written by other developers or provided by a third-party. Even if the author of the `increment` directive knew that Ajax updates were required, they could not be performed because all of the Ajax update logic is contained in the controller and not accessible to a directive, especially one in another module.

The solution to this problem is to make sure that changes to the local data automatically cause the required Ajax requests to be generated, but this means that any component that needs to work with the data has to know whether the data needs to be synchronized with a remote server *and* know how to make the required Ajax requests to perform updates.

AngularJS offers a partial solution to this problem through the `$resource` service, which makes it easier to work with RESTful data in an application by hiding away the details of the Ajax requests and URL formats. I'll show you how to apply the `$resource` service in the sections that follow.

Installing the ngResource Module

The `$resource` service is defined within an optional module called `ngResource` that must be downloaded into the `angularjs` folder. Go to `http://angularjs.org`, click Download, select the version you require (version 1.2.5 is the latest version as I write this), and click the Extras link in the bottom-left corner of the window, as shown in Figure 21-7.

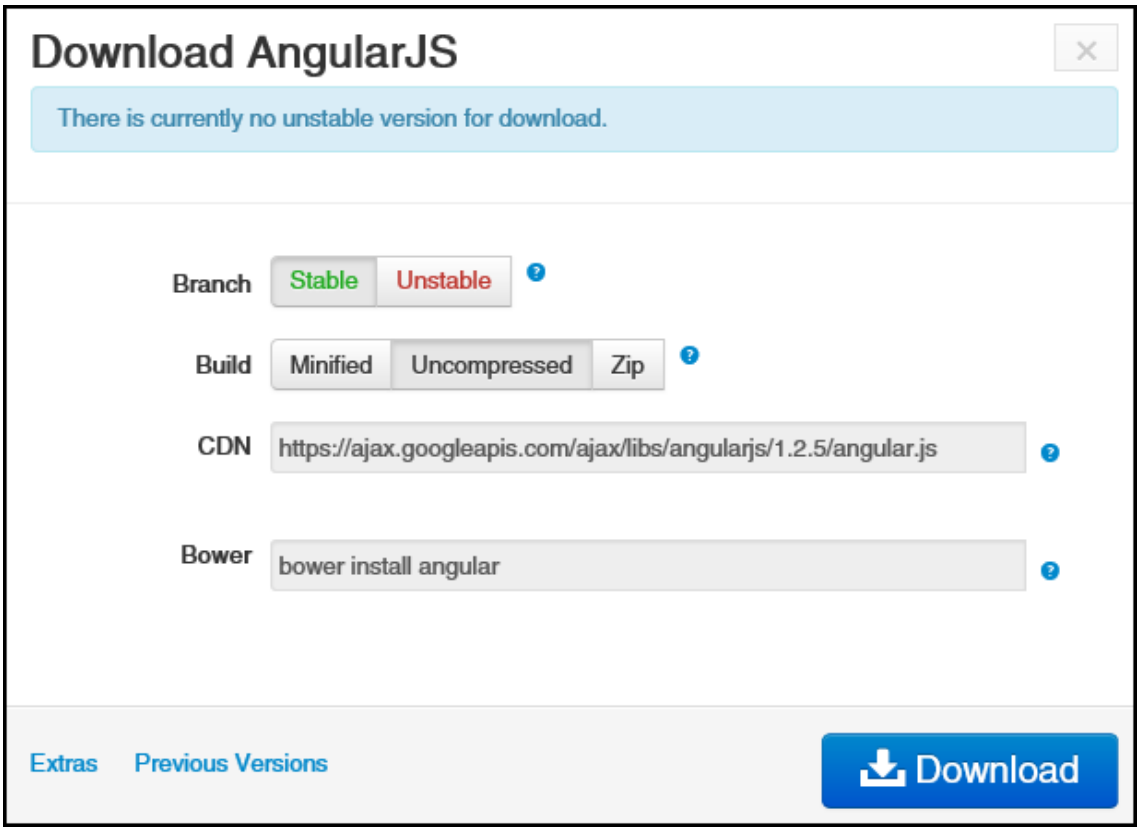

Figure 21-7. Downloading an optional module

Download the `angular-resource.js` file into the `angularjs` folder. In Listing 21-13, you can see how I have added a `script` element for the new file to the `products.html` file.

Listing 21-13. Adding a Reference to the products.html File

```
...
<head>
    <title>Products</title>
    <script src="angular.js"></script>
    <script src="angular-resource.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="products.js"></script>
    <script src="increment.js"></script>
```

```
</head>
...
```

Using the $resource Service

In Listing 21-14, you can see how I have used the `$resource` service in in the `products.js` file to manage the data that I get from the server without directly creating Ajax requests.

Listing 21-14. Using the $resource Service in the products.js File

```
angular.module("exampleApp", ["increment", "ngResource"])
.constant("baseUrl", "https://api.parse.com/1/classes/Products/")
.config(function ($httpProvider) {
    $httpProvider.defaults.headers.common["X-Parse-Application-Id"]
        = "7uRKj9WGCnHTZ136SAbh7N3TBI3NyWrpgWTgomjp";
    $httpProvider.defaults.headers.common["X-Parse-REST-API-Key"]
        = "aeZgyAPTIKbs6YAsCNyySuORW53xeKrPC6DUUqSw";
})
.controller("defaultCtrl", function ($scope, $http, $resource, baseUrl) {

    $scope.displayMode = "list";
    $scope.currentProduct = null;

    $scope.productsResource = $resource(baseUrl + ":id", { id: "@objectId" }, {
        query: {
            method: "GET", isArray: true, transformResponse: function (data, headers) {
                return JSON.parse(data).results;
            }
        },
        update: {method: "PUT"}
    });

    $scope.listProducts = function () {
        $scope.products = $scope.productsResource.query();
    }

    $scope.deleteProduct = function (product) {
        product.$delete().then(function () {
            $scope.products.splice($scope.products.indexOf(product), 1);
        })
    }

    $scope.createProduct = function (product) {
        var newProduct = new $scope.productsResource(product);
        newProduct.$save().then(function (response) {
            $scope.products.push(angular.extend(newProduct, product));
            $scope.displayMode = "list";
        });
    }

    $scope.updateProduct = function (product) {
        angular.copy(product).$update().then(function () {
            $scope.displayMode = "list";
        });
    }
```

```
    $scope.editOrCreateProduct = function (product) {
        $scope.currentProduct = product || {};
        $scope.displayMode = "edit";
    }

    $scope.saveEdit = function (product) {
        if (angular.isDefined(product.objectId)) {
            $scope.updateProduct(product);
        } else {
            $scope.createProduct(product);
        }
    }

    $scope.cancelEdit = function () {
        if ($scope.currentProduct && $scope.currentProduct.$get) {
            $scope.currentProduct.$get();
        }
        $scope.currentProduct = {};
        $scope.displayMode = "list";
    }

    $scope.listProducts();
});
```

The function signature for the behaviors defined by the controller have remained the same, which is good because it means I don't have to change any of the HTML elements in order to use the `$resource` service. The implementation of every behavior has changed, not only because the way that I obtain the data has changed but also because the assumptions that can be made about the nature of the data are different. There is a lot going on in this listing, and the `$resource` service can be confusing, so I am going to break down what's going on step-by-step in the sections that follow.

Configuring the $resource Service

The first thing I have to do is set up the `$resource` service so that it knows how to work with the RESTful Parse.com service. Here is the statement that does this:

```
...
$scope.productsResource = $resource(baseUrl + ":id", { id: "@objectId" }, {
    query: {
        method: "GET", isArray: true, transformResponse: function (data, headers) {
            return JSON.parse(data).results;
        }
    },
    update: { method: "PUT" }
});
...
```

The `$resource` service object is a function that is used to describe the URLs that are used to consume the RESTful service. The URL segments that change per object are prefixed with a colon (the `:` character). There is only one variable part of the URL I need to use, and that is the value of the `objectId` property of the `product` object, which is required when deleting or

modifying an object. For the first argument I combine the value of the `baseUrl` constant with `:id` to indicate a URL segment that will change, producing a combined value of the following:

```
https://api.parse.com/1/classes/Products/:id
```

The second argument is a configuration object whose properties specify where the value for the variable segment will come from. Each property must correspond to a variable segment from the first argument, and the value can be fixed or, as I have done in this example, bound to a property on the data object by prefixing a property name with the `@` character. I use the configuration object to specify that the value for the `id` URL segment should be taken from the `objectId` property.

Tip Most real applications will need multiple segment parts to express more complex data collections. The URL passed to the `$resource` service can contain as many variable parts as you require.

The final argument is a configuration object that defines actions that can be performed on RESTful objects or modifies the default actions. The default AngularJS REST configuration doesn't quite match the way that Parse.com works – I explain the changes I made using the configuration object in the *Configuring the $resource Service Actions* section.

The result from calling the `$resource` service function is an *access object* that can be used to query and modify the server data using the methods that I have described in Table 21-6.

Tip The `delete` and `remove` methods are identical and can be used interchangeably.

Table 21-6. *The Default Actions Defined by an Access Object*

Name	HTTP	URL	Description
`delete(params, product)`	`DELETE`	`/1/classes/Products/<id>`	Removes the object with the specified ID
`get(id)`	`GET`	`/1/classes/Products</id>`	Gets the (single) object with the specified ID
`query()`	`GET`	`/1/classes/Products`	Gets all of the objects as an array
`remove(params, product)`	`DELETE`	`/1/classes/Products</id>`	Removes the object with the specified ID
`save(product)`	`POST`	`/1/classes/Products</id>`	Saves modifications to the

object with the specified ID

Don't worry if you don't understand the role of actions at the moment; it will become clear soon.

Listing the REST Data

I assigned the access object returned from invoking the `$resource` service object to a variable called `productResource`, which I then use to get the initial snapshot of data from the server. Here is the definition of the `listProducts` behavior:

```
...
$scope.listProducts = function () {
    $scope.products = $scope.productsResource.query();
}
...
```

The access object provides me with the means to query and modify data on the server, but it doesn't automatically perform any of these actions itself, which is why I call the `query` method to get the initial data for the application. The `query` method requests the `/1/classes/Products` URL provided by the Parse.com service to get all of the data objects available.

The result from the `query` method is a *collection* array that is initially empty. The `$resource` service creates the result array and then uses the `$http` service to make an Ajax request. When the Ajax request completes, the data that is obtained from the server is placed into the collection. This is such an important point that I am going to repeat it as a caution.

Parse.com responds to queries with a JSON object that has a `result` property that contains the array of objects. I explain how I configured the access object to automatically unwrap the data in the *Configuring the $resource Service Actions* section.

Caution The array returned by the query method is initially empty and is populated only when an asynchronous HTTP request to the server has completed.

RESPONDING TO DATA LOADING

For many applications, loading the data asynchronously works perfectly well, and the changes in the scope caused the data arrives ensure that the application responds correctly. Even though the example in this chapter is simple, it illustrates the way that many, if not most, AngularJS applications are structured: the data arrives, causing a change in the scope that refreshes the bindings and displays the data in a table.

Sometimes you need to respond more directly at the moment when the data arrives. To support this, the `$resource` service adds a `$promise` property to the collection array returned by the `query` method. The promise is resolved when the Ajax request for the data is complete. Here is an example of how you would register a success handler with the promise:

```
...
$scope.listProducts = function () {
    $scope.products = $scope.productsResource.query();
    $scope.products.$promise.then(function (data) {
        // do something with the data
    });
}
...
```

The promise is fulfilled after the result array is populated, which means you can access the data through the array or through the argument passed to the `success` function. See Chapter 20 for details of promises and how they work.

The asynchronous delivery of the data works nicely with data bindings because they automatically update when the data arrives and the collection array is populated.

Modifying Data Objects

The `query` method populates the collection array with `Resource` objects, which define all of the properties specified in the data returned by the server and some methods that allow manipulation of the data without needing to use the collections array. Table 21-7 describes the methods that `Resource` objects define (plus one addition that I added with the configuration object).

Table 21-7. *The Methods Supported by Resource Objects*

Name	Description
`$delete()`	Deletes the object from the server; equivalent to calling `$remove()`
`$get()`	Refreshes the object from the server, clearing any uncommitted local changes
`$remove()`	Deletes the object from the server; equivalent to calling `$delete()`
`$save()`	Saves the object to the server
`$update()`	Updates the object to reflect any changes. This is not a default action – see the *Configuring the $resource Service Actions* sections for details.

All of the `Resource` object methods perform asynchronous requests and return `promise` objects that you can use to receive notifications when the request completes or fails. An example can be seen in the `updateProduct` behavior, which uses the `$update` method I added to the access object when I created it:

```
...
$scope.updateProduct = function (product) {
    angular.copy(product).$update().then(function () {
```

```
        $scope.displayMode = "list";
    });
}
...
```

Tip There is usually some adjustment required to get AngularJS to work with a RESTful service. You can see an example in the `updateProduct` behavior. AngularJS expects the server response to the `$update` method to be the modified data object, but the Parse.com server only sends an acknowledgement to confirm that the update has been applied and the effect is that AngularJS replaces the `product` object with one that doesn't contain any product data. To work around this problem, I use the `angular.copy` method to create a duplicate of the `product` object and call the `$update` method on it so that AngularJS updates an object that isn't part of the `products` data array.

The `$get` method is also pretty straightforward. I used it in this example to back out from abandoned edits in the `cancelEdit` behavior, as follows:

```
...
$scope.cancelEdit = function () {
    if ($scope.currentProduct && $scope.currentProduct.$get) {
        $scope.currentProduct.$get();
    }
    $scope.currentProduct = {};
    $scope.displayMode = "list";
}
...
```

Before I call the `$get` method, I check to see that it is available for me to call and the effect is to reset the edited object to the state stored on the server. This is a different approach to editing from the one I took when using the `$http` service, where I duplicated local data in order to have a reference point to which I could return when editing was cancelled.

Note I am blithely assuming that all of my Ajax requests succeed in this example for the sake of simplicity, but you should take care to respond to errors in real projects.

Deleting Data Objects

The `$delete` and `$remove` methods generate the same requests to the server and are identical in every way. The wrinkle in their use is that they send the request to remove an object from the server but don’t remove the object from the collection array. This is a sensible approach, since the outcome of the request to the server isn’t known until the response is received and the application will be out of sync with the server if the local copy is deleted and the request subsequently returns an error.

To work around this, I have used the `promise` object that these methods return to register a callback handler that synchronizes the local data upon the successful deletion at the server in the `deleteProduct` behavior, as follows:

```
...
$scope.deleteProduct = function (product) {
    product.$delete().then(function () {
        $scope.products.splice($scope.products.indexOf(product), 1);
    })
}
...
```

Creating New Objects

Using the `new` keyword on the access object provides the means to apply the `$resource` methods to data objects so that they can be saved to the server. I use this technique in the `createProduct` behavior so that I can use the `$save` method and write new objects to the database:

```
...
$scope.createProduct = function (product) {
    var newProduct = new $scope.productsResource(product);
    newProduct.$save().then(function (response) {
        $scope.products.push(angular.extend(newProduct, product));
        $scope.displayMode = "list";
    });
}
...
```

The Parse.com server responds with the `objectId` value for the newly created object at the server. I have a choice at this point – I can use the `$get` method to retrieve the new object from the server so that I can update the `products` array with fully populated object that I can use for subsequent requests. This is what I did in Chapter 8 when I used REST to administer the application.

The alternative – and the choice I made here – is to combine the methods and properties of the object I created with the `new` keyword (and which has been updated by AngularJS so that it just contains the `objectId` value sent by the server) with the `product` object that contains the data sent to the server. Here is the statement that combines both objects and updates the `products` array:

```
...
$scope.products.push(angular.extend(newProduct, product));
...
```

Caution This technique should only be used when the server doesn't add any additional properties to the data object when it is being created – or, if it does, that you don't need them. Although the approach that I used in Chapter 8 makes an additional HTTP request, it has the benefit of ensuring that you are working with complete data.

Configuring the $resource Service Actions

The `get`, `save`, `query`, `remove`, and `delete` methods that are available on the collection array and the `$`-prefixed equivalents on individual `Resource` objects are known as *actions*. By default, the `$resource` service defines the first four actions I described in Table 21-7, but these are easily configured and extended so that the methods correspond to the API provided by the server. Here is the statement that I used to create the access object:

```
...
$scope.productsResource = $resource(baseUrl + ":id", { id: "@objectId" }, {
    query: {
        method: "GET", isArray: true, transformResponse: function (data, headers) {
            return JSON.parse(data).results;
        }
    },
    update: { method: "PUT" }
});
...
```

The configuration object that is final argument to the `$resource` method adapts and extends the default AngularJS behavior to work with Parse.com. I used this feature to change the behavior of the `query` action and define a new action called `update`.

The actions are expressed as object properties whose names correspond to the action that is being defined or modified. Each action property is set to a configuration object that defines the properties shown in Table 21-8.

Table 21-8. *The Configuration Properties Used for Actions*

Name	Description
`method`	Sets the HTTP method that will be used for the Ajax request. The default method is POST.
`params`	Specifies values for the segment variables in the URL passed as the first argument to the `$resource` service function.
`url`	Overrides the default URL for this action.
`isArray`	When `true`, specifies that the response will be a JSON data array. The default value, `false`, specifies that the response to the request will be, at most, one object.
`transformRequest`	Specifies an interceptor function for requests
`transformResponse`	Specifies an interceptor function for responses

Some RESTful web services allow objects to be created and updated using the HTTP POST method but others – including Parse.com – will only accept updates with the PUT method. The action I defined – called `update` – uses the `method` configuration property to specify the HTTP method but otherwise accepts the default values:

```
...
```

```
update: { method: "PUT" }
...
```

The redefinition of the `query` action is more complex. AngularJS expects to receive an array of data objects from the `query` action, but Parse.com wraps that array up in an object as the value of a property called `results`. In order to adapt AngularJS to work with Parse.com, I give AngularJS access to the array. I do this by using the `transformResponse` configuration property to intercept the response, de-serialize the JSON sent by the server and return the value of the `results` property:

```
...
function (data, headers) {
    return JSON.parse(data).results;
}
...
```

I have to set values for the `method` and `isArray` properties when redefining the query to override the defaults.

Tip In addition, you can use the following properties to configure the Ajax request that the action will generate (I described the effect of these options in Chapter 20): `cache`, `timeout`, `withCredentials`, `responseType`, and `interceptor`.

Actions that are defined in this way are just like the defaults and can be called on the collection array and on individual `Resource` objects, as demonstrated by the `updateProduct` method, which uses the `$update` action.

Creating $resource-Ready Components

Using the `$resource` service lets me write components that can operate on RESTful data without needing to know the details of the Ajax requests that are required to manipulate the data. In Listing 21-17, you can see how I have updated the `increment` directive from earlier in the chapter so that it can be configured to use data obtained from the `$resource` service.

AVOIDING THE ASYNCHRONOUS DATA TRAP

The `$resource` service provides a partial solution to disseminating RESTful data throughout an application: It hides the details of the Ajax requests, but it still requires that the components that use the data know that the data is RESTful and should be manipulated with methods like `$save` and `$delete`.

At this point, you might be thinking of ways of completing the process and using scope watchers and event handlers to create a wrapper around the RESTful data that monitors for changes and automatically writes changes to the server.

Don't be tempted to try this. It is a trap, and it doesn't—in fact, it *can't*—ever work properly because you will be trying to hide the asynchronous nature of the Ajax requests that underpin REST from the components that use the data. Code that doesn't know that RESTful data is being used will assume that all operations take effect immediately and that the data in the browser is the authoritative reference, neither of which is true when there are Ajax requests being fired off in the background.

Things fall apart completely when the server returns an error, which will reach the browser long after the synchronous operation on the data has completed and execution of the code has moved on. There is no compelling way of dealing with errors: You can't unwind the operation without risking causing an inconsistent state application (because execution of the synchronous code has continued), and you lack the means to signal the original code so that it can try again (because that would require awareness of the Ajax requests). The best thing you can do is dump the application state and reload the data from the server, which will come as a nasty surprise to the user.

Instead, accept that components have to be rewritten or adapted to understand the methods that the `$resource` service adds to data objects and, as I demonstrate in the updated `increment` directive, make the use of these methods configurable.

Listing 21-17. Working with RESTful Data in the increment.js File

```
angular.module("increment", [])
    .directive("increment", function () {
        return {
            restrict: "E",
            scope: {
                item: "=item",
                property: "@propertyName",
                restful: "@restful",
                method: "@methodName"
            },
            link: function (scope, element, attrs) {
                var button = angular.element("<button>").text("+");
                button.addClass("btn btn-primary btn-xs");
                element.append(button);
                button.on("click", function () {
                    scope.$apply(function () {
                        scope.item[scope.property]++;
                        if (scope.restful) {
                            angular.copy(scope.item)[scope.method]();
                        }
                    })
                })
            },
        }
    });
```

When creating components that may operate on data provided by the `$resource` service, you need to provide configuration options not only to enable the RESTful support but also to specify the action method or methods that are required to update the server. In this example, I use the value of an attribute called `restful` to configure the REST support and `method` to get the name of

the method that should be called when the value is incremented. In Listing 21-18, you can see how I apply these changes in the `tableView.html` file.

Listing 21-18. Adding Configuration Attributes in the tableView.html File

```
<div class="panel-body">
    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th>
                <th>Category</th>
                <th class="text-right">Price</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="item in products">
                <td>{{item.name}}</td>
                <td>{{item.category}}</td>
                <td class="text-right">{{item.price | currency}}</td>
                <td class="text-center">
                    <button class="btn btn-xs btn-primary"
                            ng-click="deleteProduct(item)">
                        Delete
                    </button>
                    <button class="btn btn-xs btn-primary"
                            ng-click="editOrCreateProduct(item)">
                        Edit
                    </button>
                    <increment item="item" property-name="price" restful="true"
                        method-name="$update" />
                </td>
            </tr>
        </tbody>
    </table>
    <div>
        <button class="btn btn-primary" ng-click="listProducts()">Refresh</button>
        <button class="btn btn-primary" ng-click="editOrCreateProduct()">New</button>
    </div>
</div>
```

The result is that when you click the + button in a table row, the local value is updated, and the `$update` method is then called to send the update to the server.

Summary

In this chapter I showed you how to work with RESTful services. I showed you how to manually form the Ajax requests using the `$http` service and explained why this can cause problems when the data is used beyond the component that creates it. I demonstrated how the `$resource` service can be used to hide the details of the Ajax requests, and I gave a stern warning about the dangers of trying to hide the asynchronous nature of RESTful data from the components that operate on it. In the next chapter, I describe the service that provides URL routing.

CHAPTER 22

Services for Views

In this chapter, I describe the set of services that AngularJS provides for working with views. I introduced views in Chapter 10 and showed you how to use the `ng-include` directive to import them into an application. In this chapter, I demonstrate how to use *URL routing*, which uses views to enable sophisticated navigation within an application. URL routing can be a difficult topic to understand, so I introduce the functionality gradually in this chapter, slowly revising the example application to introduce individual features. Table 22-1 summarizes this chapter.

Table 22-1. *Chapter Summary*

Problem	Solution	Listing
Enable navigation within the application.	Define URL routes using the `$routeProvider`.	1–4
Display the view from the active route.	Apply the `ng-view` directive.	5
Change the active view.	Use the `$location.path` method or use an `a` element whose `href` attribute matches the route path.	6–7
Pass information via the path.	Use route parameters in the route URL. Access the parameters using the `$routeParams` service.	8–10
Associate a controller with the view displayed by the active route.	Use the `controller` configuration property.	11
Define dependencies for the controller.	Use the `resolve` configuration property.	12–13

Why and When to Use the View Services

The services I describe in this chapter are useful for simplifying complex applications by allowing multiple components to control the content that the user sees. You won't need these services in small or simple applications.

Preparing the Example Project

For this chapter, I am going to continue to work with the example I created in Chapter 21 to demonstrate the different ways in which AngularJS applications can consume RESTful APIs. In the previous chapter, the focus was on managing the Ajax for the RESTful data, so you may not have noticed a rather nasty hack, which I will explain before showing how to resolve it.

Understanding the Problem

The application contains two view files, `tableView.html` and `editorView.html`, which I imported into the main `products.html` file using the `ng-include` directive.

The `tableView.html` file contains the default view for the application and lists the data from the server in a `table` element. I switch to the contents of the `editorView.html` file when the user is creating a new product or editing an existing one. When the operation is complete—or cancelled—I return to the contents of the `tableView.html` file again. The problem is the way that I manage the visibility of the contents of the view files. Listing 22-1 shows the `products.html` file.

Listing 22-1. The Contents of the products.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Products</title>
    <script src="angular.js"></script>
    <script src="angular-resource.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="products.js"></script>
    <script src="increment.js"></script>
</head>
<body ng-controller="defaultCtrl">
    <div class="panel panel-primary">
        <h3 class="panel-heading">Products</h3>
        <ng-include src="'tableView.html'" ng-show="displayMode == 'list'"></ng-include>
        <ng-include src="'editorView.html'" ng-show="displayMode == 'edit'"></ng-include>
    </div>
</body>
</html>
```

The issue is the use of the `ng-show` directive to control the visibility of the elements. To work out whether the contents of the view should be shown to the user, I check the value of a scope variable called `displayMode` and compare it to a literal value, like this:

```
...
<ng-include src="'tableView.html'" ng-show="displayMode == 'list'"></ng-include>
...
```

I set the value of `displayMode` in the controller behaviors defined in the `products.js` file to display the content I require. Listing 22-2 highlights how I set `displayMode` in the `products.js` file to switch between the views.

Listing 22-2. Setting the displayMode Value in the products.js File

```
angular.module("exampleApp", ["increment", "ngResource"])
.constant("baseUrl", "https://api.parse.com/1/classes/Products/")
.config(function ($httpProvider) {
    $httpProvider.defaults.headers.common["X-Parse-Application-Id"]
        = "7uRKj9WGCnHTZ136SAbh7N3TBI3NyWrpgWTgomjp";
    $httpProvider.defaults.headers.common["X-Parse-REST-API-Key"]
        = "aeZgyAPTIKbs6YAsCNyySuORW53xeKrPC6DUUqSw";
})
.controller("defaultCtrl", function ($scope, $http, $resource, baseUrl) {

    $scope.displayMode = "list";
    $scope.currentProduct = null;

    $scope.productsResource = $resource(baseUrl + ":id", { id: "@objectId" }, {
        query: {
            method: "GET", isArray: true, transformResponse: function (data, headers) {
                return JSON.parse(data).results;
            }
        },
        update: { method: "PUT" }
    });

    $scope.listProducts = function () {
        $scope.products = $scope.productsResource.query();
    }

    $scope.deleteProduct = function (product) {
        product.$delete().then(function () {
            $scope.products.splice($scope.products.indexOf(product), 1);
        })
    }

    $scope.createProduct = function (product) {
        var newProduct = new $scope.productsResource(product);
        newProduct.$save().then(function (response) {
            $scope.products.push(angular.extend(newProduct, product));
            $scope.displayMode = "list";
        });
    }

    $scope.updateProduct = function (product) {
        angular.copy(product).$update().then(function () {
            $scope.displayMode = "list";
        });
    }

    $scope.editOrCreateProduct = function (product) {
        $scope.currentProduct = product || {};
        $scope.displayMode = "edit";
    }

    $scope.saveEdit = function (product) {
        if (angular.isDefined(product.objectId)) {
            $scope.updateProduct(product);
        } else {
```

```
            $scope.createProduct(product);
        }
    }

    $scope.cancelEdit = function () {
        if ($scope.currentProduct && $scope.currentProduct.$get) {
            $scope.currentProduct.$get();
        }
        $scope.currentProduct = {};
        $scope.displayMode = "list";
    }

    $scope.listProducts();
});
```

This approach works, but it presents a problem, which is that any component that needs to change the layout of the application needs access to the `displayMode` variable, which is assigned to the controller scope. This isn't too much of a burden in such a simple application where the view is always managed by a single controller, but it doesn't scale up when additional components need to control what the user sees.

What's needed is a way to separate the view selection from the controller so that the application content can be driven from any part of the application, and that's what I will show you in this chapter.

Using URL Routing

AngularJS supports a feature called *URL routing*, which uses the value returned by the `$location.path` method to load and display view files without the need for nasty literal values embedded throughout the markup and code in an application. In the sections that follow, I'll show you how to install and use the `$route` service, which provides the URL routing functionality.

Installing the ngRoute Module

The `$route` service is defined within an optional module called `ngRoute` that must be downloaded into the `angularjs` folder. Go to `http://angularjs.org`, click Download, select the version you require (version 1.2.5 is the latest version as I write this), and click the Extras link in the bottom-left corner of the window, as shown in Figure 22-1.

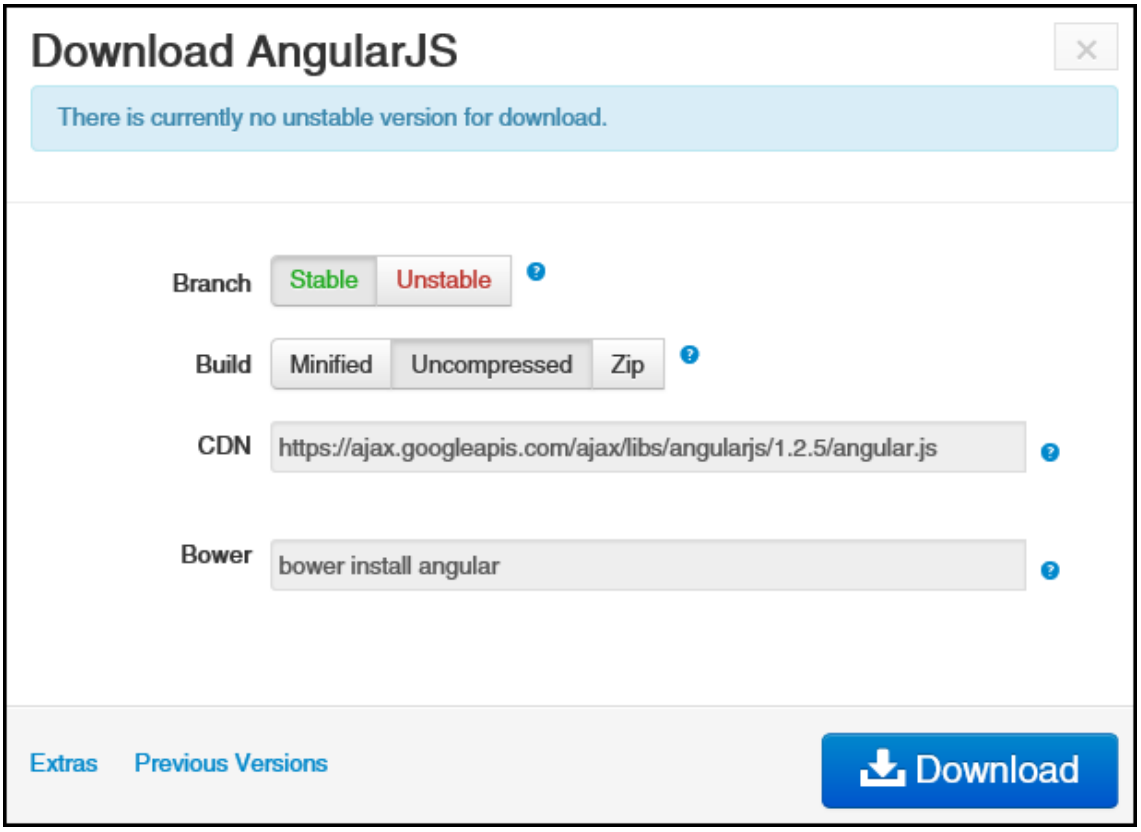

Figure 22-1. Downloading an optional module

Download the `angular-route.js` file into the `angularjs` folder. In Listing 22-3, you can see how I have added a `script` element for the new file to the `products.html` file.

Listing 22-3. Adding a Reference to the products.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Products</title>
    <script src="angular.js"></script>
    <script src="angular-resource.js"></script>
    <script src="angular-route.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="products.js"></script>
    <script src="increment.js"></script>
</head>
<body ng-controller="defaultCtrl">
    <div class="panel panel-primary">
        <h3 class="panel-heading">Products</h3>
        <ng-include src="'tableView.html'" ng-show="displayMode == 'list'"></ng-include>
        <ng-include src="'editorView.html'" ng-show="displayMode == 'edit'"></ng-include>
```

```
    </div>
</body>
</html>
```

Defining the URL Routes

At the heart of the functionality provided by the `$route` service is a set of mappings between URLs and view file names, known as *URL routes* or just *routes*. When the value returned by the `$location.path` method matches one of the mappings, the corresponding view file will be loaded and displayed. The mappings are defined using the provider for the `$route` service, `$routeProvider`. Listing 22-4 shows how I have defined routes for the example application.

Listing 22-4. Defining Routes in the product.js File

```
angular.module("exampleApp", ["increment", "ngResource", "ngRoute"])
.constant("baseUrl", "https://api.parse.com/1/classes/Products/")
.config(function ($routeProvider, $locationProvider) {

    $locationProvider.html5Mode(true);

    $routeProvider.when("/list", {
        templateUrl: "/tableView.html"
    });

    $routeProvider.when("/edit", {
        templateUrl: "/editorView.html"
    });

    $routeProvider.when("/create", {
        templateUrl: "/editorView.html"
    });

    $routeProvider.otherwise({
        templateUrl: "/tableView.html"
    });

})
.config(function ($httpProvider) {
    $httpProvider.defaults.headers.common["X-Parse-Application-Id"]
...
```

I have added a dependency on the `ngRoute` module and added a `config` function to define the routes. My `config` function declares dependencies on providers for the `$route` and `$location` services, the latter of which I use to enable HTML5 URLs.

Tip I am going to use HTML5 URLs in this chapter because they are cleaner and simpler and I know that the browser I will be using supports the HTML5 History API. See Chapter 19 for details of the `$location` service support for HTML5, how to detect that the browser provides the required features, and the potential for problems.

Routes are defined using the `$routeProvider.when` method. The first argument is the URL that the route will apply to, and the second argument is the route configuration object. The routes I have defined are the simplest possible because the URLs are static and I have provided the minimum configuration information, but later in the chapter I'll show you more complex examples. I'll describe all of the configuration options later in the chapter, but for now it is enough to know that the `templateUrl` configuration option specifies the view file that should be used when the path of the current browser URL matches the first argument passed to the `when` method.

Tip Always specify the value of the `templateUrl` with a leading `/` character. If you do not, the URL will be evaluated relative to the value returned by the `$location.path` method, and changing this value is one of the key activities required when using routing. Without the `/` character, you will quickly generate a `Not Found` error as you navigate within the application.

The `otherwise` method is used to define a route that is used when no other one matches the current URL path. It is good practice to provide such a fallback route, and I have summarized the overall effect of the routes I have defined in Table 22-2.

Tip I didn't really need to define the route for `/list` since the route defined with the `otherwise` method displays the `tableView.html` view if no other route matches the current path. I like to be explicit when defining routes because they can become quite complex, and anything that makes them easier to read and understand is worth doing.

Table 22-2. *The Effect of the Routes Defined in the products.js File*

URL Path	View File
`/list`	`tableView.html`
`/edit`	`editorView.html`
`/create`	`editorView.html`
All other URLs	`tableView.html`

Displaying the Selected View

The `ngRoute` module includes a directive called `ng-view` that displays the contents of the view file specified by the route that matches the current URL path returned by the `$location` service. In

Listing 22-5, you can see how I am able to use the `ng-view` directive to replace the troublesome elements in the `products.html` file, removing the literal values that I dislike so much.

Listing 22-5. Using the ng-view Directive in the products.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Products</title>
    <script src="angular.js"></script>
    <script src="angular-resource.js"></script>
    <script src="angular-route.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="products.js"></script>
    <script src="increment.js"></script>
</head>
<body ng-controller="defaultCtrl">
    <div class="panel panel-primary">
        <h3 class="panel-heading">Products</h3>
        <div ng-view></div>
    </div>
</body>
</html>
```

When the value returned by the `$location/path` changes, the `$route` service evaluates the routes defined through its provider and changes the content of the element to which the `ng-view` directive has been applied.

Wiring Up the Code and Markup

All that remains is to update the code and the markup to change the URL rather than the `displayMode` variable to change the layout of the application. In JavaScript code, this means I need to use the `path` method provided by the `$location` service, as shown in Listing 22-6.

Listing 22-6. Using the $location Service to Change Views in the products.js File

```
angular.module("exampleApp", ["increment", "ngResource", "ngRoute"])
.constant("baseUrl", "https://api.parse.com/1/classes/Products/")
.config(function ($routeProvider, $locationProvider) {

    $locationProvider.html5Mode(true);

    $routeProvider.when("/list", {
        templateUrl: "/tableView.html"
    });

    $routeProvider.when("/edit", {
        templateUrl: "/editorView.html"
    });

    $routeProvider.when("/create", {
        templateUrl: "/editorView.html"
    });
```

```
    $routeProvider.otherwise({
        templateUrl: "/tableView.html"
    });

})
.config(function ($httpProvider) {
    $httpProvider.defaults.headers.common["X-Parse-Application-Id"]
        = "7uRKj9WGCnHTZ136SAbh7N3TBI3NyWrpgWTgomjp";
    $httpProvider.defaults.headers.common["X-Parse-REST-API-Key"]
        = "aeZgyAPTIKbs6YAsCNyySuoRW53xeKrPC6DUUqSw";
})
.controller("defaultCtrl", function ($scope, $http, $resource, $location, baseUrl) {

    $scope.currentProduct = null;

    $scope.productsResource = $resource(baseUrl + ":id", { id: "@objectId" }, {
        query: {
            method: "GET", isArray: true, transformResponse: function (data, headers) {
                return JSON.parse(data).results;
            }
        },
        update: { method: "PUT" }
    });

    $scope.listProducts = function () {
        $scope.products = $scope.productsResource.query();
    }

    $scope.deleteProduct = function (product) {
        product.$delete().then(function () {
            $scope.products.splice($scope.products.indexOf(product), 1);
        })
    }

    $scope.createProduct = function (product) {
        var newProduct = new $scope.productsResource(product);
        newProduct.$save().then(function (response) {
            $scope.products.push(angular.extend(newProduct, product));
            $location.path("/list");
        });
    }

    $scope.updateProduct = function (product) {
        angular.copy(product).$update().then(function () {
            $location.path("/list");
        });
    }

    $scope.editOrCreateProduct = function (product) {
        $scope.currentProduct = product || {};
        $location.path("/edit");
    }

    $scope.saveEdit = function (product) {
        if (angular.isDefined(product.objectId)) {
            $scope.updateProduct(product);
```

```
        } else {
            $scope.createProduct(product);
        }
        $scope.currentProduct = {};
    }

    $scope.cancelEdit = function () {
        if ($scope.currentProduct && $scope.currentProduct.$get) {
            $scope.currentProduct.$get();
        }
        $scope.currentProduct = {};
        $location.path("/list");
    }

    $scope.listProducts();
});
```

This isn't a huge change. I have added a dependency on the `$location` service and replaced the calls that changed the `displayMode` value with equivalent calls to the `$location.path` method. There is a more interesting change, however: I replaced the `editOrCreateProduct` behavior with one called `editProduct`, which is slightly simpler. Here is the old behavior:

```
...
$scope.editOrCreateProduct = function (product) {
    $scope.currentProduct = product ? product : {};
    $scope.displayMode = "edit";
}
...
```

And here is its replacement:

```
...
$scope.editProduct = function (product) {
    $scope.currentProduct = product;
    $location.path("/edit");
}
...
```

The old behavior was the start point for both the editing and creation process, which were differentiated by the `product` argument. If the `product` argument wasn't `null`, then I used the object to set the `currentProduct` variable, which populates the fields in the `editorView.html` view.

Tip There is one other change highlighted in the listing. I have updated the `saveEdit` behavior to reset the value of the `currentProduct` variable. Without this change, the values from an edit operation are displayed to the user if they subsequently create a new product. This is a temporary problem that will be resolved as I expand the support for routing in the application.

The reason I am able to simplify the behavior is that the routing feature allows me to initiate the process of creating a new product object just by changing the URL. In Listing 22-7, you can see the changes I have made to the `tableView.html` file.

Listing 22-7. Adding Support for Routes to the tableView.html File

```
<div class="panel-body">
    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th>
                <th>Category</th>
                <th class="text-right">Price</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="item in products">
                <td>{{item.name}}</td>
                <td>{{item.category}}</td>
                <td class="text-right">{{item.price | currency}}</td>
                <td class="text-center">
                    <button class="btn btn-xs btn-primary"
                            ng-click="deleteProduct(item)">
                        Delete
                    </button>
                    <button class="btn btn-xs btn-primary" ng-click="editProduct(item)">
                        Edit
                    </button>
                    <increment item="item" property-name="price" restful="true"
                               method-name="$update" />
                </td>
            </tr>
        </tbody>
    </table>
    <div>
        <button class="btn btn-primary" ng-click="listProducts()">Refresh</button>
        <a href="create" class="btn btn-primary">New</a>
    </div>
</div>
```

I have replaced the `button` element whose `ng-click` directive invoked the old behavior and replaced it with an `a` element whose `href` attribute specifies the URL that matches the route that displays the `editorView.html` view. Bootstrap allows me to style `button` and `a` elements to look the same, so there is no discernable difference in the layout to the user. However, when the `a` element is clicked, the URL changes to `/create` and the `editorView.html` view is displayed, as shown in Figure 22-2.

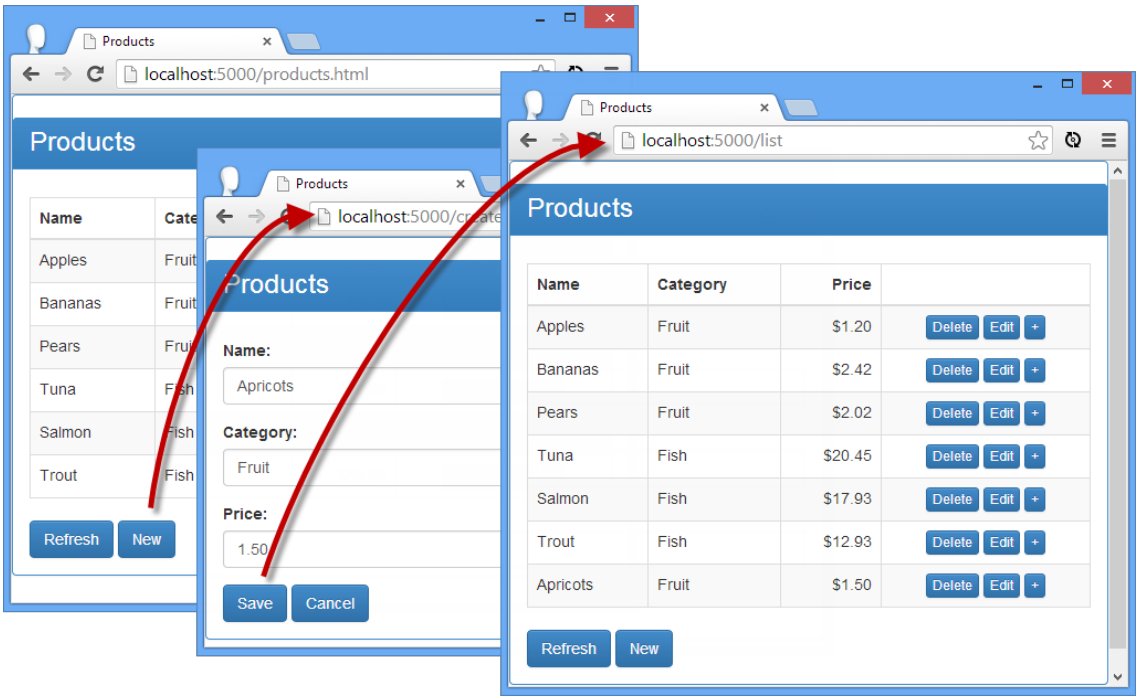

Figure 22-2. Navigating within the application

To see the effect, load the `products.html` file into the browser and click the New button. The URL displayed by the browser will change from `http://localhost:5000/products.html` to `http://localhost:5000/create`. This is the magic of HTML5 URLs managed through the HTML5 Browser History API, and the contents of the `editorView.html` view will be displayed. Enter details of a new product and click the Save button (or Cancel if you prefer), and the contents of the `tableView.html` view are shown again, with a URL of `http://localhost:5000/list`.

Caution Routing works when the application changes the URL, but it doesn't work if the user changes it; the browser takes any URL that the user enters as being a literal request for a file and tries to request the corresponding content from the server.

Using Route Parameters

The URLs I used to define the routes in the previous section were *fixed* or *static*, meaning that the value passed to the `$location.path` method or set in an `a` element's `href` attribute has to exactly match the value I used with the `$routeProvider.when` method. As a reminder, here is one of the routes that I defined:

```
...
$routeProvider.when("/create", {
```

```
    templateUrl: "editorView.html"
});
...
```

This route will be activated only when the path component of the URL matches `/create`. This is the most basic kind of URL that routes can be used with and, as a consequence, the most limited.

Route URLs can contain *route parameters*, which match one or more *segments* in the path displayed by the browser. A segment is the set of characters between two `/` characters. As an example, the segments in the URL `http://localhost:5000/users/adam/details` are `users`, `adam`, and `details`. There are two kinds of route parameters: *conservative* and *eager*. A conservative route parameter will match one segment, and an eager one will match as many segments as possible. To demonstrate how this works, I have changed the routes in the `products.js` file, as shown in Listing 22-8.

Listing 22-8. Defining Routes with Route Parameters in the products.js File

```
...
.config(function ($routeProvider, $locationProvider) {

    $locationProvider.html5Mode(true);

    $routeProvider.when("/list", {
        templateUrl: "/tableView.html"
    });

    $routeProvider.when("/edit/:id", {
        templateUrl: "/editorView.html"
    });

    $routeProvider.when("/edit/:id/:data*", {
        templateUrl: "/editorView.html"
    });

    $routeProvider.when("/create", {
        templateUrl: "/editorView.html"
    });

    $routeProvider.otherwise({
        templateUrl: "/tableView.html"
    });
})
...
```

The first highlighted route URL, `/edit/:id`, contains a conservative route parameter. The variable is denoted by a colon character (`:`) and then a name, which is `id` in this case. The route will match a path such as `/edit/1234`, and it will assign the value of `1234` to a route parameter called `id`. (Route variables are accessed through the `$routeParams` service, which I describe shortly.)

Routes that use only static segments and conservative route parameters will match only those paths that contain the same number of segments as their URL. In the case of the `/edit/:id`

URL, only URLs that contain two segments where the first segment is `edit` will be matched. Paths with more or less segments won't match and nor will paths whose first segment isn't `edit`.

You can extend the range of paths that a route URL will match by including an eager route parameter, like this:

```
...
$routeProvider.when("/edit/:id/:data*", {
...
```

An eager route parameter is denoted by a colon, followed by a name, followed by an asterisk. The example will match any path that has at least three segments where the first segment is `edit`. The second segment will be assigned to the route parameter `id`, and the remaining segments will be assigned to the route parameter `data`.

Tip Don't worry if segment variables and route parameters don't make sense at the moment. You will see how they work as I develop the examples in the following sections.

Accessing Routes and Routes Parameters

The URLs I used in the previous section process paths and assign the contents of segments to route parameters, which can then be accessed in code. In this section, I am going to demonstrate how to access those values using the `$route` and `$routeParams` services, both of which are contained in the `ngRoute` module. My first step is to change the button that edits product objects in the `tableView.html` file, as shown in Listing 22-9.

Listing 22-9. Using Routing to Trigger Editing in the tableView.html File

```
<div class="panel-body">
    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th>
                <th>Category</th>
                <th class="text-right">Price</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="item in products">
                <td>{{item.name}}</td>
                <td>{{item.category}}</td>
                <td class="text-right">{{item.price | currency}}</td>
                <td class="text-center">
                    <button class="btn btn-xs btn-primary"
                            ng-click="deleteProduct(item)">
                        Delete
                    </button>
                    <a href="/edit/{{item.objectId}}"
                        class="btn btn-xs btn-primary">Edit</a>
```

```
                <increment item="item" property-name="price" restful="true"
                           method-name="$update" />
            </td>
        </tr>
    </tbody>
</table>
<div>
    <button class="btn btn-primary" ng-click="listProducts()">Refresh</button>
    <a href="create" class="btn btn-primary">New</a>
</div>
</div>
```

I have replaced the `button` element with an `a` element whose `href` element corresponds to one of the routing URLs I defined in Listing 22-9, which I achieve using a standard inline binding expression within the `ng-repeat` directive. This means that each row in the `table` element will contain an `a` element like this one:

```
<a href="/edit/18d5f4716c6b1acf" class="btn btn-xs btn-primary">Edit</a>
```

When this link is clicked, the route parameter called `id` that I defined in Listing 22-8 will be assigned the value `18d5f4716c6b1acf`, which corresponds to the `id` property of the product object that the user wants to edit. In Listing 22-10, you can see that I have updated the controller in the `products.js` file to respond to this change.

Listing 22-10. Accessing a Route Parameter in the products.js File

```
...
.controller("defaultCtrl", function ($scope, $http, $resource, $location,
    $route, $routeParams, baseUrl) {

    $scope.currentProduct = null;


    $scope.$on("$routeChangeSuccess", function () {
        if ($location.path().indexOf("/edit/") == 0) {
            var id = $routeParams["id"];
            for (var i = 0; i < $scope.products.length; i++) {
                if ($scope.products[i].objectId == id) {
                    $scope.currentProduct = $scope.products[i];
                    break;
                }
            }
        }
    });

    $scope.productsResource = $resource(baseUrl + ":id", { id: "@objectId" }, {
        query: {
            method: "GET", isArray: true, transformResponse: function (data, headers) {
                return JSON.parse(data).results;
            }
        },
        update: { method: "PUT" }
    });
```

```
    $scope.listProducts = function () {
        $scope.products = $scope.productsResource.query();
    }

    $scope.deleteProduct = function (product) {
        product.$delete().then(function () {
            $scope.products.splice($scope.products.indexOf(product), 1);
        })
    }

    $scope.createProduct = function (product) {
        var newProduct = new $scope.productsResource(product);
        newProduct.$save().then(function (response) {
            $scope.products.push(angular.extend(newProduct, product));
            $location.path("/list");
        });
    }

    $scope.updateProduct = function (product) {
        angular.copy(product).$update().then(function () {
            $location.path("/list");
        });
    }

    $scope.editProduct = function (product) {
        $scope.currentProduct = product;
        $location.path("/edit");
    }

    $scope.saveEdit = function (product) {
        if (angular.isDefined(product.objectId)) {
            $scope.updateProduct(product);
        } else {
            $scope.createProduct(product);
        }
        $scope.currentProduct = {};
    }

    $scope.cancelEdit = function () {
        if ($scope.currentProduct && $scope.currentProduct.$get) {
            $scope.currentProduct.$get();
        }
        $scope.currentProduct = {};
        $location.path("/list");
    }

    $scope.listProducts();
});
```

There is a lot going on in the highlighted code, so I am going to break down each major part and explain them in turn in the sections that follow.

Note I have removed the `editProduct` behavior from the controller, which was previously invoked to initiate the editing process and displayed the `editorView.html` view. The behavior is no longer required since editing is not initiated through the routing system.

Responding to Route Changes

The `$route` service, for which I added a dependency in Listing 22-10, can be used to manage the currently selected route. Table 22-3 shows the methods and properties that the `$route` service defines.

Table 22-3. *The Methods and Properties Defined by the $route Service*

Name	Description
`current`	Returns an object that provides information about the active route. The object returned from this property defines a `controller` property that returns the controller associated with the route (see the "Using Controllers with Routes" section) and a `locals` property that provides the set of controller dependencies (see the "Adding Dependencies to Routes" section). The collection returned by the `locals` property also contains `$scope` and `$template` properties that return the scope for the controller and the view content.
`reload()`	Reloads the view even if the URL path hasn't changed.
`routes`	Returns the collections of the routes defined through the `$routeProvider`.

I didn't use any of the members described in Table 22-3, but I did rely on another aspect of the `$route` service, which is a set of events used to signal changes in the active route, as described in Table 22-4. Handlers for these methods are registered using the scope `$on` method, which I described in Chapter 15.

Table 22-4. *The Events Defined by the $route Service*

Name	Description
`$routeChangeStart`	Triggered before the route is changed
`$routeChangeSuccess`	Triggered after the route has changed
`$routeUpdate`	Triggered when the route is refreshed; this is tied to the `reloadOnSearch` configuration property, which I describe in the "Configuring Routes" section
`$routeChangeError`	Triggered if the route cannot be changed

Most of the `$route` service isn't that useful. You usually need to know about two things: when the route changes and what the new path is. The `$routeChangeSuccess` method provides the

first piece of information, and the `$location` service (not `$route`) provides the second, as demonstrated by this fragment that repeats the key statements from the `products.js` file:

```
...
$scope.$on("$routeChangeSuccess", function () {
    if ($location.path().indexOf("/edit/") == 0) {
        // ...statements for responding to /edit route go here...
    }
});
...
```

I register a handler function that is invoked when the current route changes, and I use the `$location.path` method to figure out what state the application is in. If the path starts with `/edit/`, then I know I have to respond to an edit operation.

Getting the Route Parameters

When dealing with a path that starts with `/edit/`, I know I need to get the value of the `id` route parameter so that I can populate the fields of the `editorView.html` file. Route parameter values are accessed through the `$routeParams` service. The parameter values are presented as a collection that is indexed by name, as follows:

```
...
$scope.$on("$routeChangeSuccess", function () {
    if ($location.path().indexOf("/edit/") == 0) {
        var id = $routeParams["id"];
        for (var i = 0; i < $scope.products.length; i++) {
            if ($scope.products[i].objectId == id) {
                $scope.currentProduct = $scope.products[i];
                break;
            }
        }
    }
});
...
```

I obtain the value of the `id` parameter and then use it to locate the object that the user wants to edit.

Caution For simplicity in this example, I assume that the value of the `id` route parameter is in the right format and corresponds to the `objectId` value of an object in the data array. You should be more careful in a real project and validate the values you receive.

Configuring Routes

The routes I have defined so far in this chapter have defined only the `templateUrl` configuration property, which specifies the URL of the view file that the route will display. This is only one of

the configuration options available. I have listed the full set in Table 22-5, and I describe the two most important, `controller` and `resolve`, in the sections that follow.

Table 22-5. *The Route Configuration Options*

Name	Description
`controller`	Specifies the name of a controller to be associated with the view displayed by the route. See the "Using Controllers with Routes" section.
`controllerAs`	Specifies an alias to be used for the controller.
`template`	Specifies the content of the view. This can be expressed as a literal HTML string or as a function that returns the HTML.
`templateUrl`	Specifies the URL of the view file to display when the route matches. This can be expressed as a string or as a function that returns a string.
`resolve`	Specifies a set of dependencies for the controller. See the "Adding Dependencies to Routes" section.
`redirectTo`	Specifies a path that the browser should be redirected to when the route is matched. Can be expressed as a string or a function.
`reloadOnSearch`	When `true`, the default value, the route will reload when only the values returned by the `$location` `search` and `hash` methods change.
`caseInsensitiveMatch`	When `true`, the default value, routes are matched to URLs without case sensitivity (e.g., `/Edit` and `/edit` are considered to be the same).

Using Controllers with Routes

If you have lots of views in an application, having them share a single controller (as I have been doing so far in this chapter) becomes unwieldy to manage and test. The `controller` configuration option allows you to specify a controller that has been registered through the `Module.controller` method for the view. The effect is to separate out the controller logic that is unique to each view, as shown in Listing 22-11.

Listing 22-11. Using a Controller with a View in the products.js File

```
angular.module("exampleApp", ["increment", "ngResource", "ngRoute"])
.constant("baseUrl", "https://api.parse.com/1/classes/Products/")
.config(function ($routeProvider, $locationProvider) {

    $locationProvider.html5Mode(true);

    $routeProvider.when("/edit/:id", {
        templateUrl: "/editorView.html",
        controller: "editCtrl"
    });
```

```
        $routeProvider.when("/create", {
            templateUrl: "/editorView.html",
            controller: "editCtrl"
        });

        $routeProvider.otherwise({
            templateUrl: "/tableView.html"
        });

    })
    .config(function ($httpProvider) {
        $httpProvider.defaults.headers.common["X-Parse-Application-Id"]
            = "7uRKj9WGCnHTZ136SAbh7N3TBI3NyWrpgWTgomjp";
        $httpProvider.defaults.headers.common["X-Parse-REST-API-Key"]
            = "aeZgyAPTIKbs6YAsCNyySuORW53xeKrPC6DUUqSw";
    })
    .controller("defaultCtrl", function ($scope, $http, $resource, $location,
        $route, $routeParams, baseUrl) {

        $scope.currentProduct = null;

        $scope.$on("$routeChangeSuccess", function () {
            if ($location.path().indexOf("/edit/") == 0) {
                var id = $routeParams["id"];
                for (var i = 0; i < $scope.products.length; i++) {
                    if ($scope.products[i].objectId == id) {
                        $scope.currentProduct = $scope.products[i];
                        break;
                    }
                }
            }
        });

        $scope.productsResource = $resource(baseUrl + ":id", { id: "@objectId" }, {
            query: {
                method: "GET", isArray: true, transformResponse: function (data, headers) {
                    return JSON.parse(data).results;
                }
            },
            update: { method: "PUT" }
        });

        $scope.listProducts = function () {
            $scope.products = $scope.productsResource.query();
        }

        $scope.deleteProduct = function (product) {
            product.$delete().then(function () {
                $scope.products.splice($scope.products.indexOf(product), 1);
            })
        }

        $scope.createProduct = function (product) {
            var newProduct = new $scope.productsResource(product);
            newProduct.$save().then(function (response) {
                $scope.products.push(angular.extend(newProduct, product));
                $location.path("/list");
```

```
        });
    }

    $scope.listProducts();
})
.controller("editCtrl", function ($scope, $routeParams, $location) {

    $scope.currentProduct = null;

    if ($location.path().indexOf("/edit/") == 0) {
        var id = $routeParams["id"];
        for (var i = 0; i < $scope.products.length; i++) {
            if ($scope.products[i].objectId == id) {
                $scope.currentProduct = $scope.products[i];
                break;
            }
        }
    }

    $scope.cancelEdit = function () {
        if ($scope.currentProduct && $scope.currentProduct.$get) {
            $scope.currentProduct.$get();
        }
        $scope.currentProduct = {};
        $location.path("/list");
    }

    $scope.updateProduct = function (product) {
        angular.copy(product).$update().then(function () {
            $location.path("/list");
        });
    }

    $scope.saveEdit = function (product) {
        if (angular.isDefined(product.objectId)) {
            $scope.updateProduct(product);
        } else {
            $scope.createProduct(product);
        }
        $scope.currentProduct = {};
    }
});
```

I have defined a new controller called `editCtrl` and moved the code from the `defaultCtrl` controller that is unique to supporting the `editorView.html` view. I then associate this controller with the routes that display the `editorView.html` file using the `controller` configuration property.

A new instance of the `editCtrl` controller will be created each time that the `editorView.html` view is displayed, which means I don't need to use the `$route` service events to know when the view has changed. I can just rely on the fact that my controller function is being executed.

One of the nice aspects of using controllers in this way is that the standard inheritance rules that I described in Chapter 13 apply, such that the `editCtrl` is nested within the `defaultCtrl` and can access the data and behaviors defined in its scope. This means I can define the common data and functionality in the top-level controller and just define the view-specific features in the nested controllers.

Adding Dependencies to Routes

The `resolve` configuration property allows you to specify dependencies that will be injected into the controller specified with the `controller` property. These dependencies can be services, but the `resolve` property is more useful for performing work required to initialize the view. This is because you can return promise objects as dependencies, and the route won't instantiate the controller until they are resolved. In Listing 22-12, you can see how I have added a new controller to the example and used the `resolve` property to load the data from the server.

Listing 22-12. Using the resolve Configuration Property in the products.js File

```
angular.module("exampleApp", ["increment", "ngResource", "ngRoute"])
.constant("baseUrl", "https://api.parse.com/1/classes/Products/")
.factory("productsResource", function ($resource, baseUrl) {

    return $resource(baseUrl + ":id", { id: "@objectId" }, {
        query: {
            method: "GET", isArray: true, transformResponse: function (data, headers) {
                return JSON.parse(data).results;
            }
        },
        update: { method: "PUT" }
    });
})
.config(function ($routeProvider, $locationProvider) {

    $locationProvider.html5Mode(true);

    $routeProvider.when("/edit/:id", {
        templateUrl: "/editorView.html",
        controller: "editCtrl"
    });

    $routeProvider.when("/create", {
        templateUrl: "/editorView.html",
        controller: "editCtrl"
    });

    $routeProvider.otherwise({
        templateUrl: "/tableView.html",
        controller: "tableCtrl",
        resolve: {
            data: function (productsResource) {
                return productsResource.query();
            }
        }
    });

})
.config(function ($httpProvider) {
    $httpProvider.defaults.headers.common["X-Parse-Application-Id"]
        = "7uRKj9WGCnHTZ136SAbh7N3TBI3NyWrpgWTgomjp";
    $httpProvider.defaults.headers.common["X-Parse-REST-API-Key"]
        = "aeZgyAPTIKbs6YAsCNyySuORW53xeKrPC6DUUqSw";
})
```

```
.controller("defaultCtrl", function ($scope, $location, $routeParams, productsResource) {

    $scope.data = {}

    $scope.currentProduct = null;

    $scope.deleteProduct = function (product) {
        product.$delete().then(function () {
            $scope.data.products.splice($scope.data.products.indexOf(product), 1);
        })
    }

    $scope.createProduct = function (product) {
        var newProduct = new productsResource(product);
        newProduct.$save().then(function (response) {
            $scope.data.products.push(angular.extend(newProduct, product));
            $location.path("/list");
        });
    }
})
.controller("tableCtrl", function ($scope, $location, $route, data) {

    $scope.data.products = data;

    $scope.refreshProducts = function () {
        $route.reload();
    }
})
.controller("editCtrl", function ($scope, $routeParams, $location) {

    $scope.currentProduct = null;

    if ($location.path().indexOf("/edit/") == 0) {
        var id = $routeParams["id"];
        for (var i = 0; i < $scope.data.products.length; i++) {
            if ($scope.data.products[i].objectId == id) {
                $scope.currentProduct = $scope.data.products[i];
                break;
            }
        }
    }

    $scope.cancelEdit = function () {
        $location.path("/list");
    }

    $scope.updateProduct = function (product) {
        angular.copy(product).$update().then(function () {
            $location.path("/list");
        });
    }

    $scope.saveEdit = function (product) {
        if (angular.isDefined(product.objectId)) {
            $scope.updateProduct(product);
        } else {
            $scope.createProduct(product);
```

```
        }
        $scope.currentProduct = {};
    }
});
```

There are a lot of changes in the listing, so I'll walk you through them in turn. The most important is the change of the definition of the `/list` route so that it uses the `controller` and `resolve` properties, like this:

```
...
$routeProvider.otherwise({
    templateUrl: "/tableView.html",
    controller: "tableCtrl",
    resolve: {
        data: function (productsResource) {
            return productsResource.query();
        }
    }
});
...
```

I have specified that the route should instantiate a controller called `tableCtrl`, and I have used the `resolve` property to create a dependency called `data`. The `data` property is set to a function that will be evaluated before the `tableCtrl` controller is created, and the result will be passed as an argument called `data`.

For this example, I use the `$resource` access object to obtain the data from the server, which means that the controller won't be instantiated until it is loaded and that, as a consequence, the `tableView.html` view won't be displayed until then either.

To be able to access the access object from the route dependency, I have to create a new service, as follows:

```
...
.factory("productsResource", function ($resource, baseUrl) {
    return $resource(baseUrl + ":id", { id: "@objectId" },
            { create: { method: "POST" }, save: { method: "PUT" } });
})
...
```

This is the same code that I used to create the `productResource` object in the controller in previous listings, just moved to a service through the `factory` method (described in Chapter 18) so that it is accessible more widely in the application.

The `tableCtrl` controller is rather simple, as follows:

```
...
.controller("tableCtrl", function ($scope, $location, $route, data) {

    $scope.data.products = data;

    $scope.refreshProducts = function () {
        $route.reload();
    }
})
...
```

I receive the product information from the server via the `data` argument and simply assign it to the `$scope.data.products` property. As I explained in the previous sections, the controller/scope inheritance rules that I described in Chapter 13 apply when using controllers with routes, so I had to add an object to which the `data` property belongs to ensure that the product data is available to all of the controllers in the applications and not just the scope belonging to the `tabelCtrl` controller.

The effect of adding the dependency in the route is that I no longer need the `listProducts` behavior, so I removed it from the `defaultCtrl` controller. That stranded the Refresh button in the `tableView.html` view without a way to force reload the data, so I defined a new behavior called `refreshProducts`, which uses the `$route.reload` method I described in Table 22-3. The final JavaScript change was to simplify the `cancelEdit` behavior, which no longer needs to reload a single product object from the server when editing is cancelled because all of the data will be refreshed when the `/list` route is activated:

```
...
$scope.cancelEdit = function () {
    $scope.currentProduct = {};
    $location.path("/list");
}
...
```

To reflect the changes in the controller, I had to update the `tableView.html` file, as shown in Listing 22-13.

Listing 22-13. Updating the tableView.html File to Reflect Controller Changes

```
<div class="panel-body">
    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Name</th>
                <th>Category</th>
                <th class="text-right">Price</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="item in data.products">
                <td>{{item.name}}</td>
                <td>{{item.category}}</td>
                <td class="text-right">{{item.price | currency}}</td>
                <td class="text-center">
                    <button class="btn btn-xs btn-primary"
                            ng-click="deleteProduct(item)">
                        Delete
                    </button>
                    <a href="/edit/{{item.objectId}}"
                        class="btn btn-xs btn-primary">Edit</a>
                    <increment item="item" property-name="price" restful="true"
                               method-name="$update" />
                </td>
            </tr>
        </tbody>
    </table>
```

```
    <div>
        <button class="btn btn-primary" ng-click="refreshProducts()">Refresh</button>
        <a href="create" class="btn btn-primary">New</a>
    </div>
</div>
```

There are two simple changes. The first is to update the `ng-repeat` directive to reflect the new data structure I put in place to deal with the scope hierarchy. The second is to update the Refresh button so that it calls the `refreshProducts` behavior instead of the defunct `listProducts`. The overall effect is that the data is obtained from the server automatically when the `/list` view is activated, which simplifies the overall code in the application.

Summary

In this chapter, I showed you the built-in services that AngularJS provides for URL routing. This is an advanced technique that is most usefully applied to make large and complex applications easier to work with. In the next chapter, I describe the services that provide support for content animation and touch events.

CHAPTER 23

Services for Animation and Touch

In this chapter, I describe the service that AngularJS provides for animating content changes in the Document Object Model (DOM) and for dealing with touch events. Table 23-1 summarizes this chapter.

Table 23-1. *Chapter Summary*

Problem	Solution	Listing
Animate content transitions.	Declare a dependency on the `ngAnimate` module, use the special naming structure to define CSS styles that contain animations or transitions, and apply the classes to one of the directives that manages content.	1–4
Detect swipe gestures.	Use the `ng-swipe-left` and `ng-swipe-right` directives.	5

Preparing the Example Project

For this chapter, I am going to continue working on the example from Chapter 22. This application obtains its data from using the RESTful API provided by Parse.com. The services that I describe in this data are not limited—or even related—to RESTful data or Ajax requests, but the application provides a convenient base for demonstrating new features.

Animating Elements

The `$animate` service allows you to provide transition effects when elements are added, removed, or moved in the DOM. The `$animate` service doesn't define any animations itself but relies on the CSS3 animation and transition features. The details of CSS3 animations and transitions are beyond the scope of this book, but I provide a full description in *The Definitive Guide to HTML5* book, which is also published by Apress.

Note Unfortunately, the nature of animations makes them impossible to show in static screenshots. To understand how they work, you will need to experience the effects they generate. But you don't have to retype the code to do this. The examples in this chapter are included in the free source code download that accompanies this book, available from `www.apress.com`.

Why and When to Use the Animation Service

Animations can be a useful means of drawing the user's attention to an important change in the layout of an application, making the transition from one state to another less jarring.

Many developers treat animations as an outlet for their frustrated artistic ambition and ladle on as many as possible. The results can be annoying, especially for the user who has to endure endless special effects every time they perform a task. For a line-of-business application, where the user could be repeating the same set of actions all day, the effect is demoralizing beyond description.

Animations should be subtle, brief, and quick. The goal is to draw the user's attention to the fact that something has changed. Use animations consistently, cautiously, and—above all—sparingly.

Installing the ngAnimation Module

The `$animation` service is defined within an optional module called `ngAnimate` that must be downloaded into the `angularjs` folder. Go to `http://angularjs.org`, click Download, select the version you require (version 1.2.5 is the latest version as I write this), and click the Extras link in the bottom-left corner of the window, as shown in Figure 23-1.

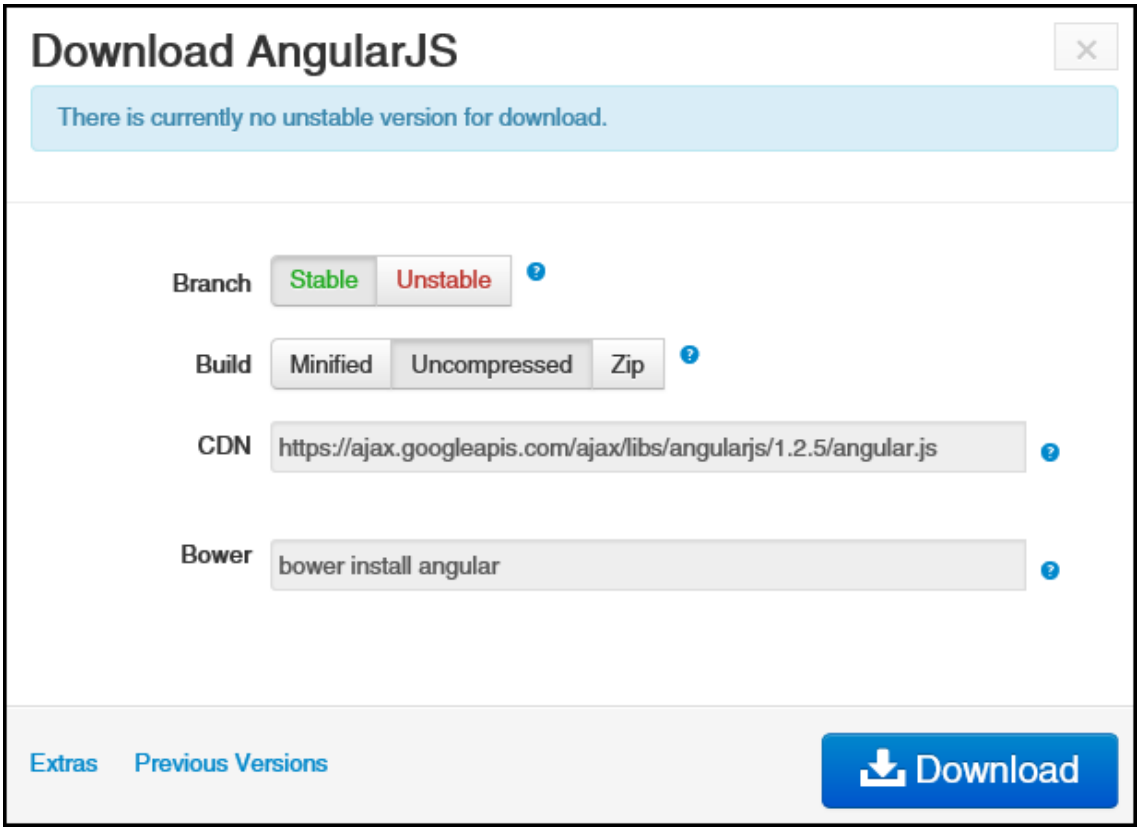

Figure 23-1. Downloading an optional module

Download the `angular-animate.js` file into the `angularjs` folder. In Listing 23-1, you can see how I have added a `script` element for the new file to the `products.html` file.

Listing 23-1. Adding a Reference to the products.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Products</title>
    <script src="angular.js"></script>
    <script src="angular-resource.js"></script>
    <script src="angular-route.js"></script>
    <script src="angular-animate.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="products.js"></script>
    <script src="increment.js"></script>
</head>
<body ng-controller="defaultCtrl">
    <div class="panel panel-primary">
        <h3 class="panel-heading">Products</h3>
        <div ng-view></div>
```

```
    </div>
</body>
</html>
```

In Listing 23-2, you can see the module dependency that I added to the `products.js` file for `ngAnimate`.

Listing 23-2. Adding the Module Dependency in the products.js File

```
angular.module("exampleApp", ["increment", "ngResource", "ngRoute", "ngAnimate"])
.constant("baseUrl", "https://api.parse.com/1/classes/Products/")
.config(function ($httpProvider) {
    $httpProvider.defaults.headers.common["X-Parse-Application-Id"]
...
```

Defining and Applying an Animation

You don't work directly with the `$animate` service to apply animations. Instead, you define animations or transitions with CSS, following a special naming convention, and then apply those names as classes to elements, which also have AngularJS directives. The best way to explain is with an example, and Listing 23-3 shows the changes I have made to the `products.html` file to animate the transition between views.

Listing 23-3. Animating View Transition in the products.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Products</title>
    <script src="angular.js"></script>
    <script src="angular-resource.js"></script>
    <script src="angular-route.js"></script>
    <script src="angular-animate.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="products.js"></script>
    <script src="increment.js"></script>
    <style type="text/css">
        .ngFade.ng-enter { transition: 0.1s linear all;  opacity: 0; }
        .ngFade.ng-enter-active { opacity: 1; }
    </style>
</head>
<body ng-controller="defaultCtrl">
    <div class="panel panel-primary">
        <h3 class="panel-heading">Products</h3>
        <div ng-view class="ngFade"></div>
    </div>
</body>
</html>
```

The key to understand what's happening in this example is the knowledge that some of the built-in directives support animations when they change their content. Table 23-2 lists directives and the names given to those changes for the purposes of animation.

The name enter is used when content is shown to the user. The name leave is used when content is hidden from the user. The name move is used when content is moved within the DOM. The names add and remove are used when content is added and removed from the DOM.

Table 23-2. *The Built-in Directives That Support Animation and the Names Associated with Them*

Directive	Names
ng-repeat	enter, leave, move
ng-view	enter, leave
ng-include	enter, leave
ng-switch	enter, leave
ng-if	enter, leave
ng-class	add, remove
ng-show	add, remove
ng-hide	add, remove

With Table 23-2 as a reference, you can get a sense of the contents of the style element I added to the example:

```
...
<style type="text/css">
    .ngFade.ng-enter { transition: 0.1s linear all;  opacity: 0; }
    .ngFade.ng-enter-active { opacity: 1; }
</style>
...
```

I have defined two CSS classes, ngFade.ng-enter and ngFade.ng-enter-active, and the names of these classes is important. The first part of the name—ngFade in this case—is the name used to apply the animations or transitions to the element, like this:

```
...
<div ng-view class="ngFade"></div>
...
```

Tip There is no requirement to prefix the top-level class name with ng, as I have done, but this is something that I have taken to doing to avoid conflicts with other CSS classes. The transition I have defined in the example causes elements to fade into view, and you might reasonably be tempted to use the name fade. However, Bootstrap, which I am also using in this example, also defines a CSS class fade, and that kind of conflict can cause problems. This has happened to me often enough that I now prefix my AngularJS animation classes with ng, just to make sure that the names are unique within the application.

The second part of the name tells AngularJS what the CSS style is to be used for. There are two names in this example: `ng-enter` and `ng-enter-active`. The `ng-` prefix *is* required, and AngularJS won't process the animation without it. The next part of the name corresponds to the details in Table 23-2. I am using the `ng-view` directive, which will perform animations when a view is displayed to the user and hidden from the user. My styles use the prefix `ng-enter`, which tells AngularJS that they should be used when a view is shown to the user.

The two styles define the start and end points for the transition that I want the `ng-view` directive to use. The `ng-enter` style defines the start point and details of the transition. I have specified that the CSS `opacity` property is initially `0` (meaning that the view is initially transparent and not visible to the user) and that the transition should be performed over a tenth of a second (I was serious when I said that animations should be brief). The `ng-enter-active` style defines the end point for the transition. I have specified that the CSS `opacity` property should be `1`, meaning that the view will be entirely opaque and so visible to the user.

The overall effect is that when the view changes, the `ng-view` directive will apply the CSS classes to the new view, which will transition it from transparent to opaque—basically, fading in the new view.

Avoiding the Perils of Parallel Animation

It is natural to assume you have to animate both the departure of old content and the arrival of new content, but doing so can be troublesome. The problem is that under normal circumstances, the `ng-view` directive adds the new view to the DOM and then removes the old one. If you try to animate the showing of the new content *and* the hiding of the old, then you will end up with both displayed at once. Listing 23-4 shows additions to the `products.html` file that will demonstrate the problem.

Listing 23-4. Adding Leave Animations to the products.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
<head>
    <title>Products</title>
    <script src="angular.js"></script>
    <script src="angular-resource.js"></script>
    <script src="angular-route.js"></script>
    <script src="angular-animate.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="products.js"></script>
    <script src="increment.js"></script>
    <style type="text/css">
        .ngFade.ng-enter { transition: 0.1s linear all;  opacity: 0; }
        .ngFade.ng-enter-active { opacity: 1; }
        .ngFade.ng-leave { transition: 0.1s linear all; opacity: 1;  }
        .ngFade.ng-leave-active { opacity: 0; }
    </style>
</head>
<body ng-controller="defaultCtrl">
    <div class="panel panel-primary">
        <h3 class="panel-heading">Products</h3>
```

```
        <div ng-view class="ngFade"></div>
    </div>
</body>
</html>
```

The result is a brief moment when both views are visible, which is unappealing and confusing to the user. The `ng-view` directive doesn't worry about trying to position views over one another, and the new content is just displayed beneath the old, as illustrated in Figure 23-2.

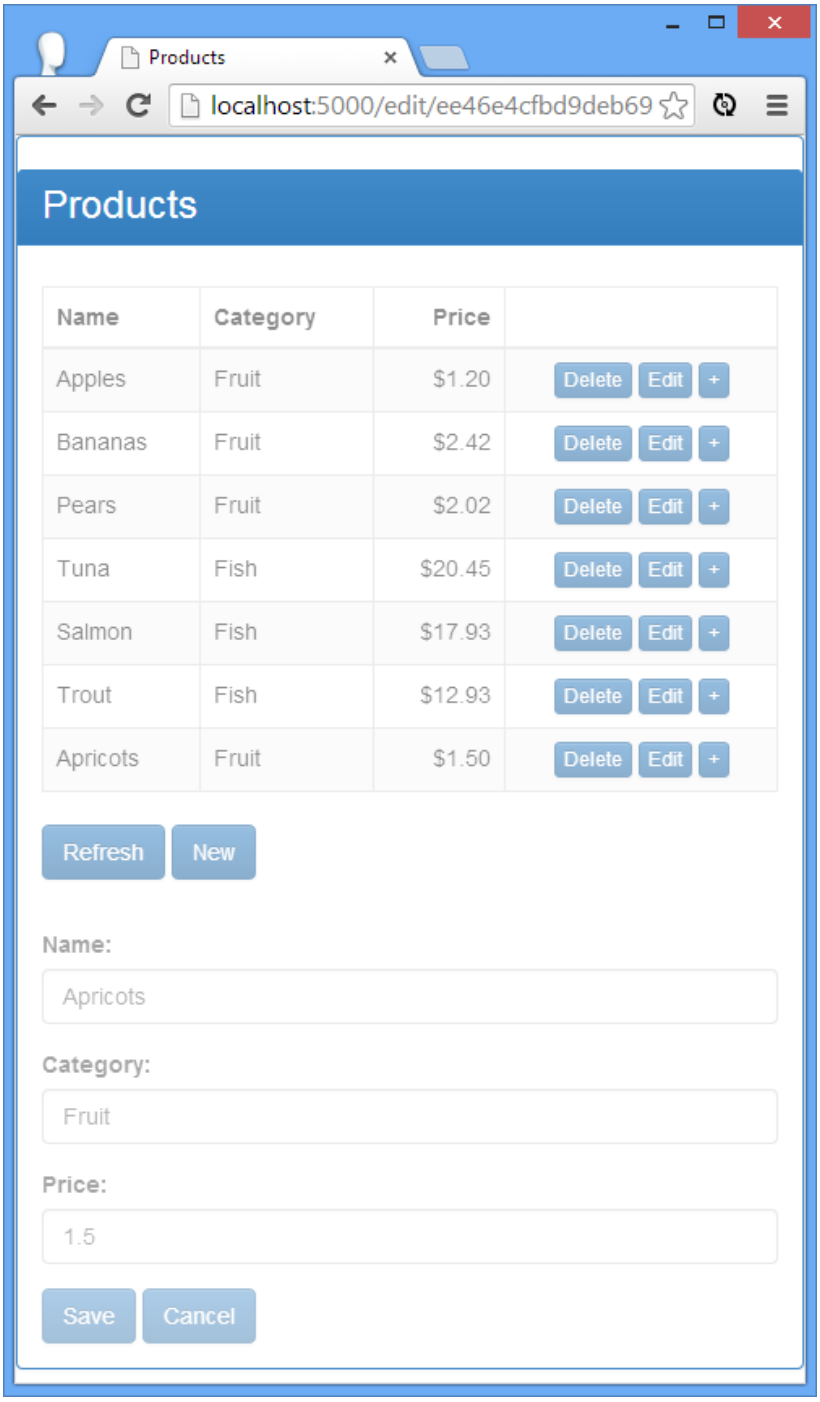

Figure 23-2. The effects of parallel animation

The content is faded because I took the screenshot at the midpoint in the transition and the `opacity` value of both views is about `0.5`. A better effect is achieved by just animating the incoming view using `enter`. It is subtle, but it makes the view transition less jarring and still draws the user's attention to the change.

Supporting Touch Events

The `ngTouch` module contains the `$swipe` service, which is used to improve support for touchscreen devices beyond the basic events I described in Chapter 11. The events in `ngTouch` module provide notification of swipe gestures and a replacement for the `ng-click` directive, which addresses a common event problem on touch-enabled devices.

Why and When to Use Touch Events

The swipe gestures are useful whenever you want to improve support for touchscreen devices. The `ngTouch` swipe events can be used to detect left-to-right and right-to-left swipe gestures. To avoid confusing the user, you must ensure that the actions you perform in response to these gestures are consistent with the rest of the underlying platform—or at the very least, the default web browser for that platform. For example, if the right-to-left gesture usually means "go back" in the web browser, then it is important that you do not interpret the gesture in your application in a different way.

The replacement for the `ng-click` directive is useful for touch-enabled browsers because they synthesize `click` events for compatibility for JavaScript code that has been written with mouse events in mind. Touch browsers generally wait for 300 milliseconds after the user has tapped the screen to see whether another tap occurs. If there is no second tap, then the browser generates the touch event to represent a `tap` *and* a `click` event to simulate a mouse—but that 300-millisecond delay is just enough of a lag to be noticeable to the user, and it can make an application appear unresponsive. The `ng-click` replacement in the `ngTouch` module doesn't wait for a second tap and issues the `click` event much faster.

Installing the ngTouch Module

The `ngTouch` module must be downloaded from `http://angularjs.org`. Follow the same procedure as for the `ngAnimate` module earlier in the chapter, but select the `angular-touch.js` file and download it into the `angularjs` folder.

Handling Swipe Gestures

To demonstrate swipe gestures, I have created an HTML file called `swipe.html` in the `angularjs` folder. Listing 23-5 shows the contents of the new file.

Listing 23-5. The Contents of the swipe.html File

```
<!DOCTYPE html>
<html ng-app="exampleApp">
```

```
<head>
    <title>Swipe Events</title>
    <script src="angular.js"></script>
    <script src="angular-touch.js"></script>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script>
        angular.module("exampleApp", ["ngTouch"])
        .controller("defaultCtrl", function ($scope, $element) {
            $scope.swipeType = "<None>";
            $scope.handleSwipe = function(direction) {
                $scope.swipeType = direction;
            }
        });
    </script>
</head>
<body ng-controller="defaultCtrl">
    <div class="panel panel-default">
        <div class="panel-body">
            <div class="well"
                 ng-swipe-right="handleSwipe('left-to-right')"
                 ng-swipe-left="handleSwipe('right-to-left')">
                <h4>Swipe Here</h4>
            </div>
            <div>Swipe was: {{swipeType}}</div>
        </div>
    </div>
</body>
</html>
```

I start by declaring a dependency on the `ngTouch` module. The event handlers are applied through the `ng-swipe-left` and `ng-swipe-right` directives. I have applied these directives to a `div` element and set them to call a controller behavior that updates a scope property that is displayed using an inline binding expression.

The swipe gestures will be detected on touch-enabled devices or when the gesture is made using the mouse. The best way to test touch events is with a touch-enabled device, of course. But if you don't have one on hand, then I find the ability of Google Chrome to simulate touch input to be useful. Click the gear icon in the bottom-right corner of the F12 tools window, select the Overrides tab, and enable the Emulate Touch Events option. Google seems to redesign the layout of the F12 tools every now and again, so you may have to hunt around to find the right option. Once touch events are enabled, you can use the mouse to swipe left and right using the mouse, and the browser will generate the required touch events, as shown in Figure 23-3.

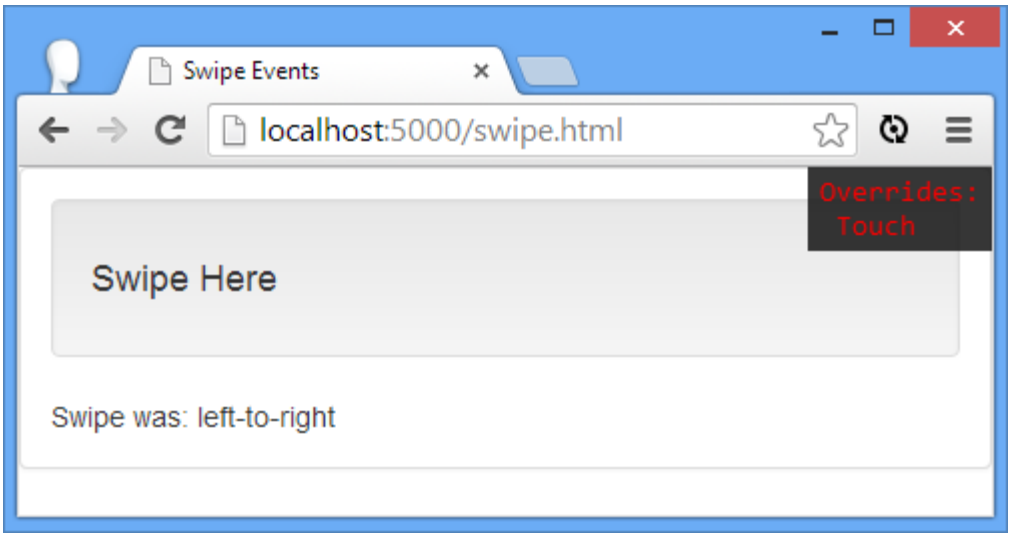

Figure 23-3. Detecting swipe gestures

Using the Replacement ng-click Directive

I am not going to demonstrate the replacement `ng-click` directive because it is a like-for-like replacement for the one I described in Chapter 11.

Summary

In this chapter, I described the services that AngularJS provides for animating elements and detecting gestures. In the next chapter, I describe some services that are used internally by AngularJS but that set the foundation for how unit testing functions.