CHAPTER 27

Web API and Single-page Applications

In this chapter, I describe the Web API feature, which is a relatively new addition to the ASP.NET platform that allows you to quickly and easily create Web services that provide an API to HTTP clients, known as *Web APIs*.

The Web API feature is based on the same foundation as the MVC Framework applications, but is not part of the MVC Framework. Instead, Microsoft has taken some key classes and characteristics that are associated with the `System.Web.Mvc` namespace and duplicated them in the `System.Web.Http` namespace. The idea is that Web API is part of the core ASP.NET platform and can be used in other types of Web applications or used as a stand-alone Web services engine. I have included Web API in this book because one of the main uses for it is to create *single-page applications* (SPAs) by combining the Web API with MVC Framework features you have seen in previous chapters. I'll explain what SPAs are and how they work later in the chapter.

That is not to take away from the way that Web API simplifies creating Web services. It is a huge improvement over the other Microsoft Web service technologies that have been appearing over the last decade or so. I like the Web API and you should use it for your projects, not least because it is simple and built on the same design that the MVC Framework uses.

I start this chapter by creating a regular MVC Framework application and then using the Web API to transform it into a single-page application. This is a surprisingly simple example, so I have treated the process like an extended example and applied some of the relevant techniques from earlier chapters because you can never have enough examples. Table 27-1 provides the summary for this chapter.

Table 27-1. *Chapter Summary*

Problem	Solution	Listing
Create a RESTful web service	Add a Web API controller to an MVC Framework application.	1–10
Map between HTTP methods and action names in a Web API controller	Apply attributes such as `HttpPut` and `HttpPost` to the methods.	11
Create a single-page application	Use Knockout and jQuery to obtain data via Ajax and bind it to HTML elements.	12–17

Understanding Single-page Applications

The term *single-page application* (SPA) is a broadly applied term. The most consistently-used definition is a web application whose initial content is delivered as a combination of HTML and JavaScript and whose subsequent operations are performed using a RESTful web service that delivers data via JSON in response to Ajax requests.

This differs from the kind of application I have been building in most of the chapters of this book, where operations performed by the user result in new HTML documents being generated in response to synchronous HTTP requests, which I will refer to as round-trip applications (RTAs).

The advantages of a SPA are that less bandwidth is required and that the user receives a smoother experience. The disadvantages are that the smoother experience can be hard to achieve and that the complexity of the JavaScript code required for a SPA demands careful design and testing.

Most applications mix and match SPA and RTA techniques, where each major functional area of the application is delivered as a SPA, and navigation between functional areas is managed using standard HTTP requests that create a new HTML document.

Preparing the Example Application

For this chapter, I created a new ASP.NET project called `WebServices` using the Empty template. I checked the options to add the folders and references for both MVC and Web API applications, as shown in Figure 27-1.

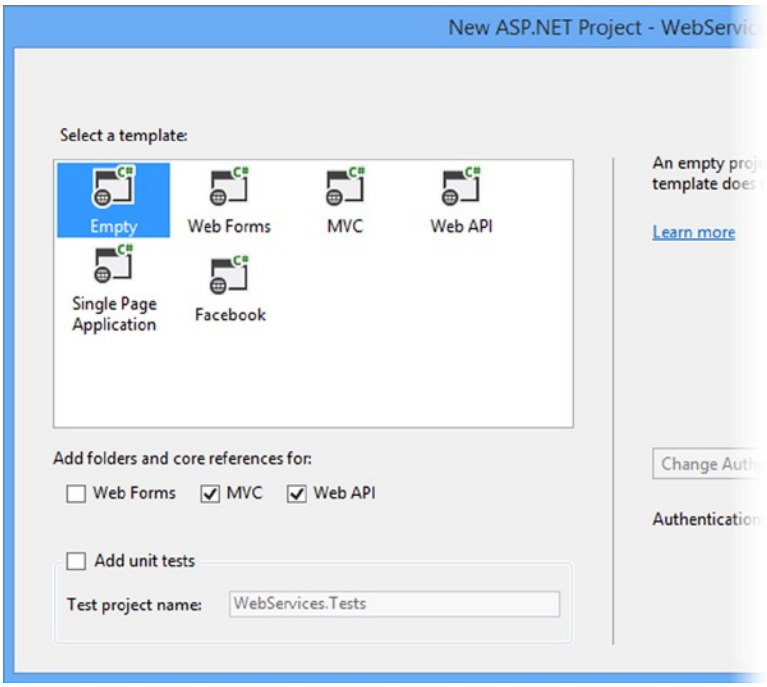

Figure 27-1. *Creating the project with the MVC and Web API references*

I will use this project to create a regular MVC Framework application and then use the Web API to create a web service. Once the web service is complete, I'll return to the MVC Framework application and make it into a single-page application.

Creating the Model

This application will create and maintain a series of reservations. I want to keep the application simple so that I can focus on the mechanics of the features I describe, and so these reservations will consist of just a name and a location. I added a class file called `Reservation.cs` to the `Models` folder, the contents of which are shown in Listing 27-1.

Listing 27-1. The Contents of the Reservation.cs File

```
namespace WebServices.Models {
    public class Reservation {
        public int ReservationId { get; set; }
        public string ClientName { get; set; }
        public string Location { get; set; }
    }
}
```

I am going to create a simple in-memory collection of `Reservation` objects to act as the model repository. I don't want to go to the trouble of setting up a database, but I do need to be able to perform CRUD operations on a collection of model objects so that I can demonstrate some important aspects of the Web API. I added a class file called `ReservationRepository.cs` to the Models folder and you can see the contents of the new file in Listing 27-2.

Listing 27-2. The Contents of the ReservationRespository.cs File

```
using System.Collections.Generic;
using System.Linq;

namespace WebServices.Models {
    public class ReservationRespository {
        private static ReservationRespository repo = new ReservationRespository();

        public static ReservationRespository Current {
            get {
                return repo;
            }
        }

        private List<Reservation> data = new List<Reservation> {
            new Reservation {
                ReservationId = 1, ClientName = "Adam", Location = "Board Room"},
            new Reservation {
                ReservationId = 2, ClientName = "Jacqui", Location = "Lecture Hall"},
            new Reservation {
                ReservationId = 3, ClientName = "Russell", Location = "Meeting Room 1"},
        };

        public IEnumerable<Reservation> GetAll() {
            return data;
        }

        public Reservation Get(int id) {
            return data.Where(r => r.ReservationId == id).FirstOrDefault();
        }

        public Reservation Add(Reservation item) {
            item.ReservationId = data.Count + 1;
            data.Add(item);
            return item;
        }
```

```
        public void Remove(int id) {
            Reservation item = Get(id);
            if (item != null) {
                data.Remove(item);
            }
        }

        public bool Update(Reservation item) {
            Reservation storedItem = Get(item.ReservationId);
            if (storedItem != null) {
                storedItem.ClientName = item.ClientName;
                storedItem.Location = item.Location;
                return true;
            } else {
                return false;
            }
        }
    }
}
```

■ **Tip** In a real project, I would be concerned about tight coupling between classes and introduce interfaces and dependency injection into the application. My focus in this chapter is just on the Web API and SPA applications, so I am going to take some shortcuts when it comes to other standard techniques.

The repository class has an initial list of three `Reservation` objects and defines methods that allow me to view, add, delete and update the collection. Since there is no persistent storage, any changes that are made to the repository will be lost when the application is stopped or restarted, but this example is all about the way in which content can be delivered and not how it is stored by the server. To ensure that there is some persistence between requests, I have created a static instance of the `ReservationRespository` class, which is accessible through the `Current` property.

Adding the NuGet Packages

I am going to rely on three NuGet packages in this chapter: jQuery, Bootstrap and Knockout. I have already described and used jQuery and Bootstrap in earlier chapters. Knockout is the library that Microsoft has adopted for single-page applications. It was created by Steve Sanderson, whom I worked with on an earlier edition of this book and who works for the Microsoft ASP.NET team. Even though Steve works for Microsoft, the Knockout package is open source and widely used and you can learn more about it at http://knockoutjs.com. I'll explain how Knockout works later in the chapter, but for the moment I just need to install the NuGet packages. Select `Package Manager Console` from the Visual Studio `Tools` ➤ `Library Package Manager` menu and enter the following commands:

```
Install-Package jquery -version 1.10.2
Install-Package bootstrap -version 3.0.0
Install-Package knockoutjs -version 3.0.0
```

Adding the Controller

I added a controller called Home to the example project, the definition of which you can see in Listing 27-3.

Listing 27-3. The Contents of the HomeController.cs File

```
using System.Web.Mvc;
using WebServices.Models;

namespace WebServices.Controllers {

    public class HomeController : Controller {
        private ReservationRespository repo = ReservationRespository.Current;

        public ViewResult Index() {
            return View(repo.GetAll());
        }

        public ActionResult Add(Reservation item) {
            if (ModelState.IsValid) {
                repo.Add(item);
                return RedirectToAction("Index");
            } else {
                return View("Index");
            }
        }

        public ActionResult Remove(int id) {
            repo.Remove(id);
            return RedirectToAction("Index");
        }

        public ActionResult Update(Reservation item) {
            if (ModelState.IsValid && repo.Update(item)) {
                return RedirectToAction("Index");
            } else {
                return View("Index");
            }
        }
    }
}
```

This is a fairly typical controller for such a simple application. Each of the action methods corresponds directly to one of the methods in the repository and the only value that the controller adds is to perform model validation, to select views, and perform redirections. In a real project, there would be more business domain logic, of course, but because the example application I am using is so basic, the controller ends up being little more than a wrapper around the repository.

Adding the Layout and Views

To generate the content for the application, I started by creating the Views/Shared folder and adding a view file called _Layout.cshtml to it, the contents of which are shown by Listing 27-4.

Listing 27-4. The Contents of the _Layout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" />
    @RenderSection("Scripts")
</head>
<body>
    @RenderSection("Body")
</body>
</html>
```

This is a basic layout that has link elements for the Bootstrap CSS files. I have defined two layout sections, Scripts and Body, that I will use to insert content into the layout. My next step was to create the top-level view for the application. Although I am going through the process of creating a regular MVC Framework application, I know that I am going to end up with a single-page application and the transformation will be made easier if I create a single view that contains all the HTML that the application will require, even if it results in an odd appearance initially. I added a view file called Index.cshtml to the Views/Home folder, the contents of which you can see in Listing 27-5.

Listing 27-5. The Contents of the Index.cshtml File

```
@using WebServices.Models

@model IEnumerable<Reservation>
@{
    ViewBag.Title = "Reservations";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Scripts {

}

@section Body {
    <div id="summary" class="section panel panel-primary">
        @Html.Partial("Summary", Model)
    </div>
```

```
    <div id="editor" class="section panel panel-primary">
        @Html.Partial("Editor", new Reservation())
    </div>
}
```

The view model for this view is an enumeration of `Reservation` objects and I rely on two partial views to provide the functional building blocks that the user will see. The first partial view file is called `Summary.cshtml`. I created the file in the `Views/Home` folder and you can see the contents of the file in Listing 27-6.

Listing 27-6. The Contents of the Summary.cshtml File

```
@model IEnumerable<WebServices.Models.Reservation>

<div class="panel-heading">Reservation Summary</div>
<div class="panel-body">
    <table class="table table-striped table-condensed">
        <thead>
            <tr><th>ID</th><th>Name</th><th>Location</th><th></th></tr>
        </thead>
        <tbody>
            @foreach (var item in Model) {
                <tr>
                    <td>@item.ReservationId</td>
                    <td>@item.ClientName</td>
                    <td>@item.Location</td>
                    <td>
                        @Html.ActionLink("Remove", "Remove",
                                new { id = item.ReservationId },
                                new { @class = "btn btn-xs btn-primary" })
                    </td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

The view model for the partial view is the same enumeration of `Reservation` object and I use it to generate a Bootstrap-styled `table` element that displays the object property values. I use the `Html.ActionLink` helper method to generate a link that will invoke the `Remove` action on the `Home` controller and use Bootstrap to style it as a button.

The other partial view is called `Editor.cshtml` and I put this in the `Views/Home` folder as well. Listing 27-7 shows the contents of this file. This partial view contains a form that can be used to create new reservations. Submitting the form invokes the `Add` action on the `Home` controller.

Listing 27-7. The Contents of the Editor.cshtml File

```
@model WebServices.Models.Reservation

<div class="panel-heading">
    Create Reservation
</div>
```

```
<div class="panel-body">
   @using(Html.BeginForm("Add", "Home")) {
        <div class="form-group">
            <label>Client Name</label>
            @Html.TextBoxFor(m => m.ClientName, new {@class = "form-control"})
        </div>
        <div class="form-group">
            <label>Location</label>
            @Html.TextBoxFor(m => m.Location, new { @class = "form-control" })
        </div>
        <button type="submit" class="btn btn-primary">Save</button>
   }
</div>
```

Setting the Start Location and Testing the Example Application

The last preparatory step is to set the location that Visual Studio will navigate to when the application is started. Select `WebServices Properties` from the Visual Studio `Project` menu, switch to the `Web` tab and check the `Specific Page` option in the `Start Action` section. You don't have to provide a value. Just checking the option is enough. To test the application in its classic MVC Framework form, select `Start Debugging` from the Visual Studio `Debug` menu. You will see the (slightly odd) all-in-one layout that provides the user with a list of the current reservations and the ability to create and delete items, as shown in Figure 27-2.

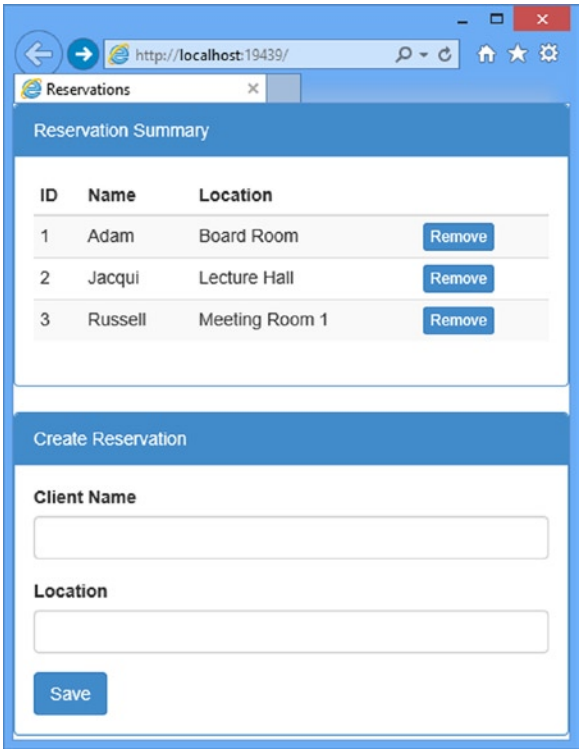

Figure 27-2. *Testing the example application*

Using Web API

The Web API feature is based on adding a special kind of controller to an MVC Framework application. This kind of controller, called an *API Controller*, has two distinctive characteristics:

1. Action methods return model, rather than `ActionResult`, objects.
2. Action methods are selected based on the HTTP method used in the request.

The model objects that are returned from an API controller action method are encoded as JSON and sent to the client. API controllers are designed to deliver Web data services, so they do not support views, layouts, or any of the other features that I used to generate HTML in the example application.

■ **Tip** The inability of an API controller to generate HTML from views is the reason that single-page applications combine standard MVC Framework techniques with the Web API. The MVC Framework performs the steps required to deliver HTML content to the user (including authentication, authorization, and selecting and rendering a view). Once the HTML is delivered to the browser, the Ajax requests generated by the JavaScript it contains are handled by the Web API controller.

As I demonstrated in Chapter 23, you can create action methods in regular controllers that return JSON data to support Ajax, but the API controller offers an alternative approach that separates the data-related actions in your application from the view-related actions, and makes creating a general-purpose Web API quick and simple.

Creating the Web API Controller

Adding Web API to an application is incredibly simple. In part this is because I am creating a basic web service, but also because the integration with the underpinnings of the MVC Framework means that little work is required. I created the `WebController.cs` class file in the `Controllers` folder of the project and used it to define the controller shown in Listing 27-8.

Listing 27-8. The Contents of the WebController.cs File

```
using System.Collections.Generic;
using System.Web.Http;
using WebServices.Models;

namespace WebServices.Controllers {

    public class WebController : ApiController {
        private ReservationRespository repo = ReservationRespository.Current;

        public IEnumerable<Reservation> GetAllReservations() {
            return repo.GetAll();
        }

        public Reservation GetReservation(int id) {
            return repo.Get(id);
        }
```

```
        public Reservation PostReservation(Reservation item) {
            return repo.Add(item);
        }

        public bool PutReservation(Reservation item) {
            return repo.Update(item);
        }

        public void DeleteReservation(int id) {
            repo.Remove(id);
        }
    }
}
```

That is all that is required to create a Web API. The API controller has a set of five action methods that map to the capabilities of the repository and provide web service access to the `Reservation` objects.

Testing the API Controller

I will explain how the API controller works shortly, but first I am going to perform a simple test. Start the application. Once the browser loads the root URL for the project, navigate to the `/api/web` URL. The result that you see will depend on the browser that you are using. If you are using Internet Explorer, then you will be prompted to save or open a file that contains the following JSON data:

```
[{"ReservationId":1,"ClientName":"Adam","Location":"Board Room"},
 {"ReservationId":2,"ClientName":"Jacqui","Location":"Lecture Hall"},
 {"ReservationId":3,"ClientName":"Russell","Location":"Meeting Room 1"}]
```

If you navigate to the same URL using a different browser, such as Google Chrome, then the browser will display the following XML data:

```
<ArrayOfReservation>
    <Reservation>
        <ClientName>Adam</ClientName>
        <Location>Board Room</Location>
        <ReservationId>1</ReservationId>
    </Reservation>
    <Reservation>
        <ClientName>Jacqui</ClientName>
        <Location>Lecture Hall</Location>
        <ReservationId>2</ReservationId>
    </Reservation>
    <Reservation>
        <ClientName>Russell</ClientName>
        <Location>Meeting Room 1</Location>
        <ReservationId>3</ReservationId>
    </Reservation>
</ArrayOfReservation>
```

There are a couple of interesting things to note here. The first is that the request for the /api/web URL has produced a list of all of the model objects and their properties, from which we can infer that the GetAllReservations action method in the Reservation controller was called.

The second point to note is that different browsers received different data formats. You might get different results if you try this yourself because later versions of the browsers may change the way they make requests, but you can see that one of requests has produced JSON and the other has produced XML. (You can also see why JSON has largely replaced XML for Web services. XML is more verbose and harder to process, especially when you are using JavaScript.)

The different data formats are used because the Web API uses the HTTP Accept header contained in the request to work out what data type the client would prefer to work with. Internet Explorer got JSON because this is the Accept header it sends:

```
...
Accept: text/html, application/xhtml+xml, */*
...
```

The browser specified that it would like text/html content most of all, and then application/xhtml+xml. The final part of the Accept header is */*, which means the browser will accept any data type if the first two are not available.

The Web API supports JSON and XML, but it gives preference to JSON, which is what it used to respond to the */* part of the IE Accept header. Here is the Accept header that Google Chrome sent:

```
...
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
...
```

I have highlighted the important part: Chrome has said that it prefers to receive application/xml data in preference to the */* catchall. The Web API controller honored this preference and delivered the XML data. I mention this because a common problem with Web API is getting an undesired data format. This happens because the Accept header gives unexpected preference to a format, or it is missing from the request entirely.

Understanding How the API Controller Works

You will understand a lot more about how the API controller works by navigating to the /api/web/3 URL. You will see the following JSON (or the equivalent XML if you are using another browser):

```
{"ReservationId":3,"ClientName":"Russell","Location":"Meeting Room 1"}
```

This time, the Web API controller has returned details of the Reservation object whose ReservationId value corresponds to the last segment of the URL I requested. The format of the URL and the use of the URL segment should remind you of Chapter 15, where I explained how MVC Framework routes work.

API controllers have their own routing configuration, which is completely separate from the rest of the application. You can see the default configuration that Visual Studio creates for new projects by looking at the /App_Start/WebApiConfig.cs file, which I have shown in Listing 27-9. This is one of the files that Visual Studio adds to the project when you check the Web API box during project creation.

Listing 27-9. The Contents of the WebApiConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

namespace WebServices {
    public static class WebApiConfig {
        public static void Register(HttpConfiguration config) {

            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

The `WebApiConfig.cs` file contains the routes used for API Controllers, but uses different classes from the regular MVC routes defined in the `RouteConfig.cs` file. The Web API feature is implemented as a stand-alone ASP.NET feature and it can be used outside of the MVC Framework, which means that Microsoft has duplicated some key MVC Framework functionality in the `System.Web.Http` namespace to keep MVC and Web API features separate. (This seems oddly duplicative when writing an MVC Framework application but makes sense since Microsoft is trying to target non-MVC developers with Web API, too.) Visual Studio also adds a call from the `Application_Start` method in the `Global.asax` class so that the Web API routes are added to the application configuration, as shown in Listing 27-10.

Listing 27-10. The Contents of the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using System.Web.Security;
using System.Web.SessionState;
using System.Web.Http;

namespace WebServices {
    public class Global : HttpApplication {
        void Application_Start(object sender, EventArgs e) {
            AreaRegistration.RegisterAllAreas();
            GlobalConfiguration.Configure(WebApiConfig.Register);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

The result is that the application has two sets of routes: those used for MVC Framework controllers and those used for Web API controllers.

Understanding API Controller Action Selection

The default Web API route, which you can see in Listing 27-9, has a static `api` segment, and `controller` and `id` segment variables, the latter being optional. They key difference from a regular MVC route is that there is no `action` segment variable, and this is where the behavior of API controllers takes shape.

When a request comes in to the application that matches a Web API route, the action is determined from the HTTP method used to make the request. When I tested the API controller by requesting `/api/reservation` using the browser, the browser specified the `GET` method.

The `ApiController` class, which is the base for API controllers, knows which controller it needs to target from the route and uses the HTTP method to look for suitable action methods.

The convention when naming API controller action methods is to prefix the name with the action method that it supports and include some reference to the model type that it operates on. But this is just a convention because Web API will match any action method whose name contains the HTTP method used to make the request.

For the example, that means that a GET request results in a choice between the `GetAllReservations` and `GetReservation`, but method names like `DoGetReservation` or just `ThisIsTheGetAction` would also be matched.

To decide between the two action methods, the controller looks at the arguments that the contenders accept and uses the routing variables to make the best match. When requesting the `/api/reservation` API, there were no routing variables except for `controller`, and so the `GetAllReservations` method was selected because it has no arguments. When requesting the `/api/reservation/3` URL, I supplied a value for the optional `id` segment variable, which made the `GetReservation` the better match because it accepts an `id` argument.

The other actions in the Web API controller are targeted using other HTTP methods: POST, DELETE, and PUT. This is the foundation for the *Representation State Transfer* (REST) style of Web API, known more commonly as a *RESTful* service, where an operation is specified by the combination of a URL and the HTTP method used to request it.

■ **Note** REST is a style of API rather than a well-defined specification, and there is disagreement about what exactly makes a Web service RESTful. One point of contention is that purists do not consider Web services that return JSON as being RESTful. Like any disagreement about an architectural pattern, the reasons for the disagreement are arbitrary and dull. I try to be pragmatic about how patterns are applied, and JSON services are RESTful as far as I am concerned.

Mapping HTTP Methods to Action Methods

I explained that the `ApiController` base class uses the HTTP method to work out which action methods to target. It is a nice approach, but it does mean that you end up with some unnatural method names that are inconsistent with conventions you might be using elsewhere. For example, the `PutReservation` method might be more naturally called `UpdateReservation`. Not only would `UpdateReservation` make the purpose of the method more obvious, but it may allow for a more direct mapping between the actions in your controller and the methods in your repository.

■ **Tip** You might be tempted to derive your repository class from `ApiController` and expose the repository methods directly as a Web API. I recommend against that and strongly suggest you create a separate controller, even as simple as the one I created in the example. At some point, the methods you want to offer via your API and the capabilities of your repository will diverge, and having a separate API controller class will make that easier to manage.

The System.Web.Http namespace contains a set of attributes that you can use to specify which HTTP methods an action should be used for. You can see how I have applied two of these attributes in Listing 27-11 to create a more natural set of method names.

Listing 27-11. Applying Attributes in the WebController.cs File

```
using System.Collections.Generic;
using System.Web.Http;
using WebServices.Models;

namespace WebServices.Controllers {

    public class WebController : ApiController {
        private ReservationRespository repo = ReservationRespository.Current;

        public IEnumerable<Reservation> GetAllReservations() {
            return repo.GetAll();
        }

        public Reservation GetReservation(int id) {
            return repo.Get(id);
        }

        [HttpPost]
        public Reservation CreateReservation(Reservation item) {
            return repo.Add(item);
        }

        [HttpPut]
        public bool UpdateReservation(Reservation item) {
            return repo.Update(item);
        }

        public void DeleteReservation(int id) {
            repo.Remove(id);
        }
    }
}
```

You can see the duplication with the MVC Framework features and the Web API. The HttpPost and HttpPut attributes that I used in Listing 27-11 have the exact same purpose as the attributes with the same name that I used in Chapter 19, but they are defined in the System.Web.Http namespace and not System.Web.Mvc. Duplication aside, the attributes work in the same way and I have ended up with more useful method names that will still work for the POST and PUT HTTP methods. (There are, of course, attributes for all of the HTTP methods, including GET, DELETE and so on.)

Using Knockout for Single-page Applications

The purpose of creating the Web API web service is to refactor the example application so that operations on the application data can be performed using Ajax requests whose JSON results will be used to update the HTML in the browser. The overall functionality of the application will be the same, but I won't be generating complete HTML documents for each interaction between the client and the server.

The transition to a single-page application puts more of a burden on the browser because I need to preserve application state at the client. I need a data model that I can update, a series of logic operations that I can perform to transform the data and a set of UI elements that allows the user to trigger those operations. In short, I need to recreate a miniature version of the MVC pattern in the browser.

The library that Microsoft has adopted for single-page applications is Knockout, which creates a JavaScript implementation of the MVC pattern (or, more accurately, the MVVM pattern which I described in Chapter 3 and is sufficiently close to the MVC pattern that I am going to treat them as the same thing). In the sections that follow, I am going to return to the MVC Framework side of the example application and apply the Knockout library to create a simple SPA.

■ **Note** Knockout does a lot more than I am going to demonstrate here and I recommend you explore the library in more depth to see what it is capable of. You can learn more at `http://knockoutjs.com` or from my *Pro JavaScript for Web Apps* book, which is published by Apress. I like Knockout, but for more complex applications I prefer AngularJS. It has a steeper learning curve, but the investment is worthwhile. You can learn more at `http://angularjs.org` or read my *Pro AngularJS* book, which—as you might have guessed by now—is also published by Apress.

Adding the JavaScript Libraries to the Layout

The first step is to add the Knockout and jQuery files to the layout so that they are available in the view. You can see the `script` element I added in Listing 27-12.

Listing 27-12. Adding the Knockout JavaScript File to the _Layout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" />
    <script src="~/Scripts/jquery-1.10.2.js"></script>
    <script src="~/Scripts/knockout-3.0.0.js"></script>
    @RenderSection("Scripts")
</head>
<body>
    @RenderSection("Body")
</body>
</html>
```

I will be using Knockout to create the MVC implementation and jQuery to handle the Ajax requests.

Implementing the Summary

The first major change I will make is to replace the Summary partial view with some inline Knockout and jQuery. You don't have to use Knockout in a single file, but I want to leave the partial views intact so you can see the difference between the standard MVC Framework way of working and the SPA techniques. In Listing 27-13, you can see the changes that I made to the Index.cstml view file.

Listing 27-13. Using Knockout and jQuery to Implement the Summary in the Index.cshtml File

```
@using WebServices.Models
@{
    ViewBag.Title = "Reservations";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Scripts {
    <script>
        var model = {
            reservations: ko.observableArray()
        };

        function sendAjaxRequest(httpMethod, callback, url) {
            $.ajax("/api/web" + (url ? "/" + url : ""), {
                type: httpMethod, success: callback
            });
        }

        function getAllItems() {
            sendAjaxRequest("GET", function(data) {
                model.reservations.removeAll();
                for (var i = 0; i < data.length; i++) {
                    model.reservations.push(data[i]);
                }
            });
        }

        function removeItem(item) {
            sendAjaxRequest("DELETE", function () {
                getAllItems();
            }, item.ReservationId);
        }

        $(document).ready(function () {
            getAllItems();
            ko.applyBindings(model);
        });
    </script>
}
```

```
@section Body {
    <div id="summary" class="section panel panel-primary">
        <div class="panel-heading">Reservation Summary</div>
        <div class="panel-body">
            <table class="table table-striped table-condensed">
                <thead>
                    <tr><th>ID</th><th>Name</th><th>Location</th><th></th></tr>
                </thead>
                <tbody data-bind="foreach: model.reservations">
                    <tr>
                        <td data-bind="text: ReservationId"></td>
                        <td data-bind="text: ClientName"></td>
                        <td data-bind="text: Location"></td>
                        <td>
                            <button class="btn btn-xs btn-primary"
                                    data-bind="click: removeItem">Remove</button>
                        </td>
                    </tr>
                </tbody>
            </table>
        </div>
    </div>
    <div id="editor" class="section panel panel-primary">
        @Html.Partial("Editor", new Reservation())
    </div>
}
```

There is a lot going on here, so I am going to break down the changes and explain each of them in turn. The overall effect of the changes, however, is that the browser uses the Web API controller to get details of the current reservations.

■ **Note** Notice that I have removed the `@model` expression from the `Index` view in Listing 27-13. I am not using the view model objects to generate the content in the view, which means that I don't need a view model. The controller is still obtaining the `Reservation` objects from the repository and passing them to the view, but I'll fix this later in the chapter.

Defining the Ajax Functions

jQuery has excellent support for making Ajax requests. To that end, I have defined a function called `sendAjaxRequest` that I will use to target methods on the Web API controller, as follows:

```
...
function sendAjaxRequest(httpMethod, callback, url) {
    $.ajax("/api/web" + (url ? "/" + url : ""), {
        type: httpMethod, success: callback
    });
}
...
```

The $.ajax function provides access to the jQuery Ajax functionality. The arguments are the URL you want to request and an object that contains configuration parameters. The sendAjaxRequest function is a wrapper around the jQuery functionality and its arguments are the HTTP method that should be used for the request (which affects the action method selected by the controller), a callback function that will be invoked when the Ajax request has succeeded and an optional URL suffix. Using the sendAjaxRequest function as a foundation, I defined functions to get all of the data available and to delete a reservation, like this:

```
...
function getAllItems() {
    sendAjaxRequest("GET", function(data) {
        model.reservations.removeAll();
        for (var i = 0; i < data.length; i++) {
            model.reservations.push(data[i]);
        }
    });
}

function removeItem(item) {
    sendAjaxRequest("DELETE", function () {
        getAllItems();
    }, item.ReservationId);
}
...
```

The getAllItems function targets the GetAllReservations controller action method and retrieves all of the reservations from the server. The removeItem function targets the DeleteReservation action method and calls the getAllItems function to refresh the data after a deletion.

Defining the Model

Underpinning the Ajax functions is the model, which I defined like this:

```
...
var model = {
    reservations: ko.observableArray()
};
...
```

Knockout works by creating *observable objects* that it monitors for changes and uses to update the HTML displayed by the browser. My model is simple. It consists of an observable array, which is just like a regular JavaScript array, but is wired up so that any changes I made are detected by Knockout. You can see how I use the model in the Ajax functions, like this:

```
...
function getAllItems() {
    sendAjaxRequest("GET", function(data) {
        model.reservations.removeAll();
        for (var i = 0; i < data.length; i++) {
            model.reservations.push(data[i]);
        }
    });
}
...
```

The two statements I highlighted are how I get the data from the server into the model. I call the `removeAll` method to remove any existing data from the observable array and then iterate through the results I get from the server with the `push` method to populate the array with the new data.

Defining the Bindings

Knockout applies changes in the data model to HTML elements via *data bindings*. Here are the most important data bindings in the `Index` view:

```
...
<tbody data-bind="foreach: model.reservations">
    <tr>
        <td data-bind="text: ReservationId"></td>
        <td data-bind="text: ClientName"></td>
        <td data-bind="text: Location"></td>
        <td>
            <button class="btn btn-xs btn-primary"
                    data-bind="click: removeItem">Remove</button>
        </td>
    </tr>
</tbody>
...
```

Knockout is expressed using the `data-bind` attribute and there is a wide range of bindings available, three of which I have used in the view. The basic format for a data-bind attribute is:

```
data-bind="type: expression"
```

The types of the three bindings in the listing are `foreach`, `text` and `click`, and I picked these three because they represent the different ways in which Knockout can be used.

The first two, the `foreach` and `text` bindings, generate HTML elements and content from the data model. When the `foreach` binding is applied to an element, Knockout generates the child elements for each object in the expression, just like the Razor `@foreach` that I was using in the partial view.

The `text` binding inserts the value of the expression as the text of the element that it is applied to. This means that when I use this binding, for example:

```
...
<td data-bind="text: ClientName"></td>
...
```

Knockout will insert the value of the `ClientName` property of the current object being processed by the `foreach` binding, which has the same effect as the Razor `@Model.ClientName` expression I used previously.

The click directive is different: it sets up an even handler for the `click` event on the element to which it has been applied. You don't have to use Knockout to set up events, of course, but the `click` binding is integrated with the other bindings and the function you specify to call when the event is triggered is passed the data object that was being processed by the `foreach` binding when the binding was applied. This is why the `removeItem` function is able to define an argument that receives a `Reservation` object (or its JavaScript representation, anyway).

Processing the Bindings

Knockout bindings are not processed automatically, which is why I included this code in the `script` element:

```
...
$(document).ready(function () {
    getAllItems();
    ko.applyBindings(model);
});
...
```

The `$(document).ready` call is a standard jQuery technique to defer execution of a function until all of the HTML elements in the document have been loaded and processed by the browser. Once that happens, I call the `getAllItems` function to load the data from the server and then the `ko.applyBindings` function to use the data model to process the `data-bind` attributes. This final call is the one that connects the data objects to the HTML elements, generates the content I require, and sets up the event handlers.

Testing the Summary Bindings

You might be wondering why I have gone to all this trouble, given that I have essentially replaced Razor expressions with their equivalent Knockout bindings. There are three important differences and to demonstrate them fully, I am going to use the browser F12 tools.

The first difference is that the model data is no longer included in the HTML that is sent to the browser. Instead, once the HTML has been processed, the browser makes an Ajax request to the Web API controller and gets the list of reservations expressed as JSON. You can see this by starting the application and using the F12 tools to monitor the requests that the browser makes (as described in Chapter 26). Figure 27-3 shows the results.

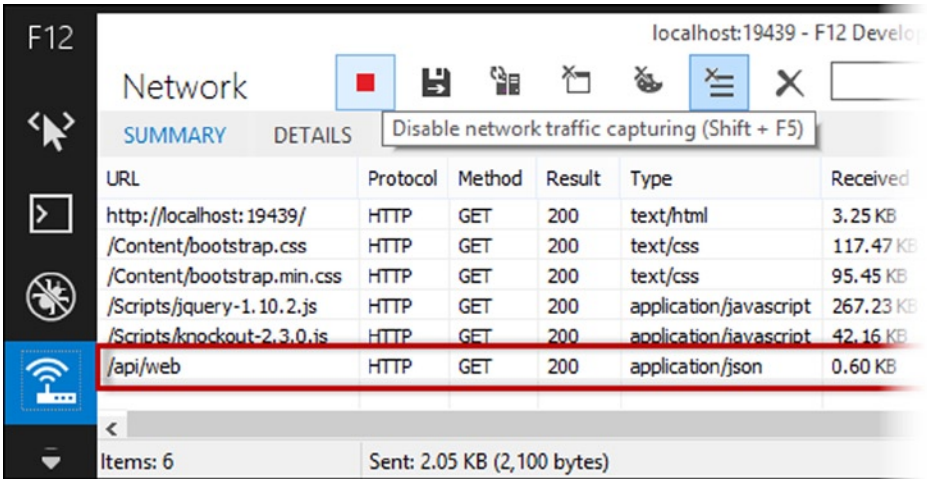

Figure 27-3. *Monitoring the connections made by the browser*

The second difference is that the data is processed by the browser, rather than at the server, as the view is rendered. To test this, you can edit the `getAllItems` function so that it doesn't make the Ajax request or process the data it receives, like this:

```
...
function getAllItems() {
    return;
    sendAjaxRequest("GET", function(data) {
        model.reservations.removeAll();
        for (var i = 0; i < data.length; i++) {
            model.reservations.push(data[i]);
        }
    });
}
...
```

The function will return before the Ajax request is made and you can see the effect by restarting the application, as shown in Figure 27-4. This may seem obvious, but it is an important characteristic of SPAs that the browser does a lot more work, including processing data and generating HTML content.

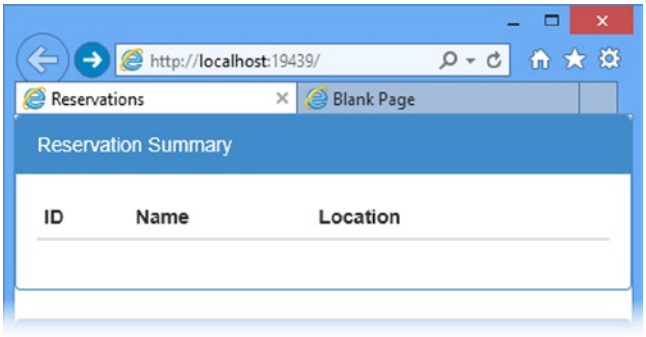

Figure 27-4. *Demonstrating that the data is retrieved and processed by the browser*

The final difference is that the data bindings are *live*, meaning that changes in the data model are reflected in the content that the `foreach` and `text` bindings generate. To test this, ensure that you return the `getAllItems` function to its working state and reload the application. Once the browser has requested, received and processed the data, open the F12 tools and switch to the `Console` section. Enter the following command into the console and hit Enter:

```
model.reservations.pop()
```

This expression removes the last item from the array of data objects in the model and as soon as you enter the command, the layout of the HTML page will reflect the change, as shown in Figure 27-5. The overall effect is that I have shifted some of the complexity of generating the HTML from the server to the client.

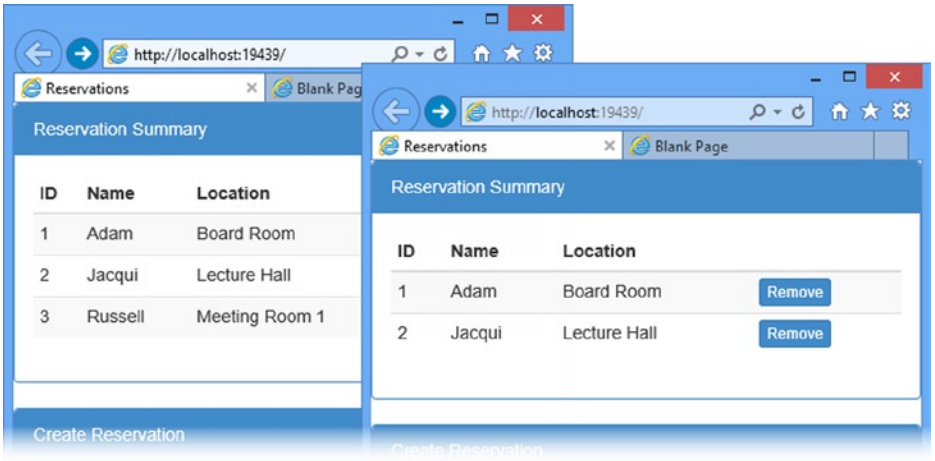

Figure 27-5. *Manipulating the model via the JavaScript console*

Improving the Delete Feature

Now that you have seen how applying Knockout has changed the nature of the client, I am going to quickly loop back and remove a shortcut that I took when I defined the Ajax methods for the application. The `removeItem` function is badly written:

```
...
function removeItem(item) {
    sendAjaxRequest("DELETE", function () {
        getAllItems();
    }, item.ReservationId);
}
...
```

I have highlighted the problem: the function makes two Ajax requests to the server—the first to perform the deletion and the second to request the contents of the repository to update the data model. Now that I have demonstrated that the client maintains its own model and that live bindings reflect model changes in the HTML, I can improve upon this function, as shown in Listing 27-14.

Listing 27-14. Improving the RemoveItem Function in the Index.cshtml File

```
...
function removeItem(item) {
    sendAjaxRequest("DELETE", function () {
        for (var i = 0; i < model.reservations().length; i++) {
            if (model.reservations()[i].ReservationId == item.ReservationId) {
                model.reservations.remove(model.reservations()[i]);
                break;
            }
        }
    }, item.ReservationId);
}
...
```

When the request to the server succeeds, I remove the corresponding data object from the model, which means that the second Ajax request is no longer required.

GETTING USED TO THE KNOCKOUT SYNTAX

There are some syntax quirks when working with Knockout observable arrays and two of them can be seen in Listing 27-14. To get an item from the array, I have to treat `model.reservations` like a function, as follows:

```
...
model.reservations()[i].ReservationId
...
```

And when it comes to removing items from the array, I use a function that is not standard JavaScript:

```
...
model.reservations.remove(model.reservations()[i]);
...
```

Knockout tries to maintain the standard JavaScript syntax, but there are some compromises required to track changes to data objects, such as these quirks. They can be confusing when you first start working with Knockout, but you soon get used to them. And you learn that when you don't get the effect you require, the likely cause is a mismatch between the standard JavaScript syntax and that required for a Knockout observable object or array. You can get further information about the Knockout API at http://knockoutjs.com.

Implementing the Create Feature

The next step is to use Knockout to replace the `Editor` partial view. Once again, I could have updated the partial view to contain the Knockout functionality, but I have chosen to include everything in the `Index.cshtml` file, as shown in Listing 27-15.

Listing 27-15. Implementing the Create Feature in the Index.cshtml File

```
@using WebServices.Models
@{
    ViewBag.Title = "Reservations";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Scripts {
    <script>
        var model = {
            reservations: ko.observableArray(),
            editor: {
                name: ko.observable(""),
                location: ko.observable("")
            }
        };
```

```
        function sendAjaxRequest(httpMethod, callback, url, reqData) {
            $.ajax("/api/web" + (url ? "/" + url : ""), {
                type: httpMethod,
                success: callback,
                data: reqData
            });
        }

        // ...other functions omitted for brevity...

        function handleEditorClick() {
            sendAjaxRequest("POST", function (newItem) {
                model.reservations.push(newItem);
            }, null, {
                ClientName: model.editor.name,
                Location: model.editor.location
            });
        }

        $(document).ready(function () {
            getAllItems();
            ko.applyBindings(model);
        });
    </script>
}

@section Body {
    <div id="summary" class="section panel panel-primary">
        <!-- elements omitted for brevity -->
    </div>
    <div id="editor" class="section panel panel-primary">
        <div class="panel-heading">
            Create Reservation
        </div>
        <div class="panel-body">
            <div class="form-group">
                <label>Client Name</label>
                <input class="form-control" data-bind="value: model.editor.name" />
            </div>
            <div class="form-group">
                <label>Location</label>
                <input class="form-control" data-bind="value: model.editor.location" />
            </div>
            <button class="btn btn-primary"
                data-bind="click: handleEditorClick">Save</button>
        </div>
    </div>
}
```

To create the editor, I have used Knockout in a different way, as I'll explain step-by-step in the sections that follow.

Extending the Model

I need to collect two pieces of information from the user in order to create a new `Reservation` in the repository: the name of the client (corresponding to the `ClientName` property) and the location (corresponding to the `Location` property). My first step is to extend the model so that I define variables that I can use to capture these values, like this:

```
...
var model = {
    reservations: ko.observableArray(),
    editor: {
        name: ko.observable(""),
        location: ko.observable("")
    }
};
...
```

The `ko.observable` function creates an observable value, which I will rely on later in the chapter. Changes to these values will be reflected in any bindings that use the `name` and `location` properties.

Implement the Input Elements

The next step is to create the `input` elements through which the user will supply values for my new model properties. I have used the Knockout `value` binding, which sets the `value` attribute on an element, as follows:

```
...
<input class="form-control" data-bind="value: model.editor.name" />
...
```

The `value` bindings ensure that the values entered by the user into the `input` elements will be used to set the model properties.

■ **Tip** Notice that I don't need a `form` element anymore. I will be using an Ajax request to send the data values to the server in response to a button click, none of which relies on the standard browser support for forms.

Creating the Event Handler

I used the `click` binding to handle the `click` event from the `button` element displayed under the `input` elements, as follows:

```
...
<button class="btn btn-primary" data-bind="click: handleEditorClick">Save</button>
...
```

The binding specifies that the handleEditorClick function should be called when the button is clicked and I defined this function in the script element, as follows:

```
...
function handleEditorClick() {
    sendAjaxRequest("POST", function (newItem) {
            model.reservations.push(newItem);
        }, null, {
            ClientName: model.editor.name,
            Location: model.editor.location
        });
}
...
```

The event handler function calls the sendAjaxRequest function. The callback adds the newly created data object sent back from the server to the model. I send an object containing the new model properties to the sendAjaxRequest function, which I have extended so that it will send them to the server as part of the Ajax request, using the data option property.

Testing the Create Feature

You can see how the Knockout implementation of the create feature works by starting the application, entering a name and location into the input elements and clicking the Save button, as illustrated by Figure 27-6.

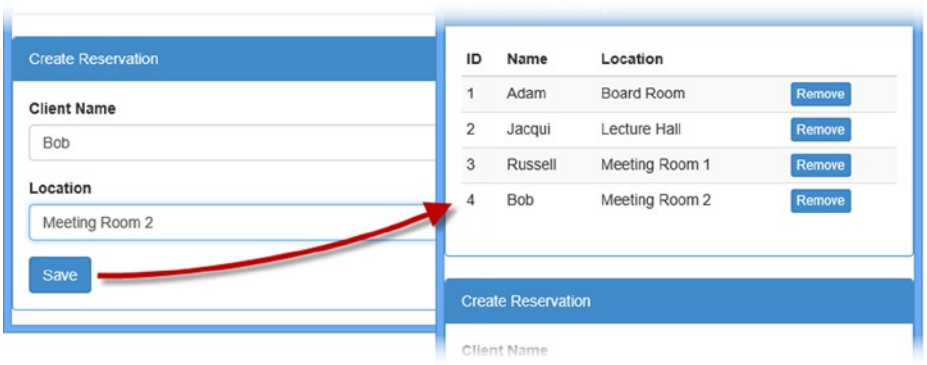

Figure 27-6. *Creating a new reservation*

Completing the Application

Now that you have seen how I can apply Knockout and the Web API to create a single-page application, I am going to finish this chapter by completing the application to add some missing features and remove some of the quirks.

Simplify the Home Controller

The Home controller is still set up with action method to manipulate the repository to retrieve and manage Reservation objects, even though all of the data displayed by the client is being requested via Ajax to the Web API controller. In Listing 27-16, you can see how I have updated the controller to remove the action methods that the Web API controller has replaced. I have also updated the Index action method so that it no longer passes a view model object.

Listing 27-16. Removing the Data Selection from the HomeController.cs File

```
using System.Web.Mvc;
using WebServices.Models;

namespace WebServices.Controllers {

    public class HomeController : Controller {

        public ViewResult Index() {
            return View();
        }
    }
}
```

Manage Content Visibility

The final change I am going to make is to manage the visibility of the elements in the HTML document so that only the summary or the editor is visible. You can see how I have done this in Listing 27-17.

Listing 27-17. Managing Element Visibility in the Index.cshtml File

```
@using WebServices.Models
@{
    ViewBag.Title = "Reservations";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Scripts {
    <script>
        var model = {
            reservations: ko.observableArray(),
            editor: {
                name: ko.observable(""),
                location: ko.observable("")
            },
            displaySummary: ko.observable(true)
        };

        function sendAjaxRequest(httpMethod, callback, url, reqData) {
            $.ajax("/api/web" + (url ? "/" + url : ""), {
                type: httpMethod,
                success: callback,
                data: reqData
            });
        }
```

```
        function getAllItems() {
            sendAjaxRequest("GET", function(data) {
                model.reservations.removeAll();
                for (var i = 0; i < data.length; i++) {
                    model.reservations.push(data[i]);
                }
            });
        }

        function removeItem(item) {
            sendAjaxRequest("DELETE", function () {
                for (var i = 0; i < model.reservations().length; i++) {
                    if (model.reservations()[i].ReservationId == item.ReservationId) {
                        model.reservations.remove(model.reservations()[i]);
                        break;
                    }
                }
            }, item.ReservationId);
        }

        function handleCreateClick() {
            model.displaySummary(false);
        }

        function handleEditorClick() {
            sendAjaxRequest("POST", function (newItem) {
                model.reservations.push(newItem);
                model.displaySummary(true);
            }, null, {
                ClientName: model.editor.name,
                Location: model.editor.location
            });
        }

        $(document).ready(function () {
            getAllItems();
            ko.applyBindings(model);
        });
    </script>
}

@section Body {
    <div id="summary" class="section panel panel-primary"
            data-bind="if: model.displaySummary">
        <div class="panel-heading">Reservation Summary</div>
        <div class="panel-body">
            <table class="table table-striped table-condensed">
                <thead>
                    <tr><th>ID</th><th>Name</th><th>Location</th><th></th></tr>
                </thead>
```

```
                <tbody data-bind="foreach: model.reservations">
                    <tr>
                        <td data-bind="text: ReservationId"></td>
                        <td data-bind="text: ClientName"></td>
                        <td data-bind="text: Location"></td>
                        <td>
                            <button class="btn btn-xs btn-primary"
                                    data-bind="click: removeItem">Remove</button>
                        </td>
                    </tr>
                </tbody>
            </table>
            <button class="btn btn-primary"
                data-bind="click: handleCreateClick">Create</button>
        </div>
    </div>
    <div id="editor" class="section panel panel-primary"
        data-bind="ifnot: model.displaySummary">
    <div class="panel-heading">
        Create Reservation
    </div>
    <div class="panel-body">
        <div class="form-group">
            <label>Client Name</label>
            <input class="form-control" data-bind="value: model.editor.name" />
        </div>
        <div class="form-group">
            <label>Location</label>
            <input class="form-control" data-bind="value: model.editor.location" />
        </div>
        <button class="btn btn-primary"
            data-bind="click: handleEditorClick">Save</button>
    </div>
</div>
}
```

I have added a property to the model that specifies whether the summary should be shown. I use this property with the `if` and `ifnot` bindings, which add and remove elements to and from the DOM based on their expression. If the `displaySummary` property is `true`, then the data summary will be shown and if it is `false`, then the editor will be shown. The final changes I made were to add a `Create` button that calls a function that changes the `displaySummary` property and an addition to the callback function that processes new items that changes it back again. You can see the final result in Figure 27-7.

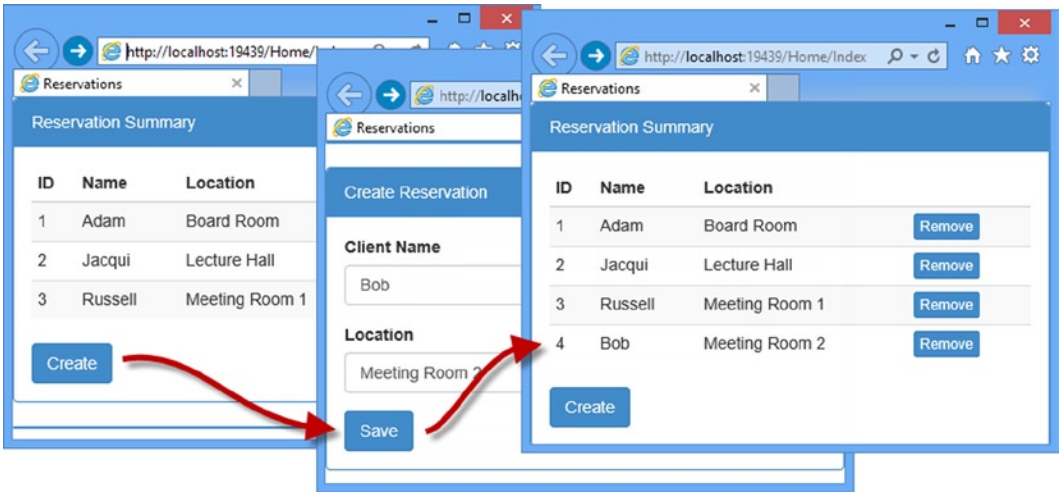

Figure 27-7. *Adding a reservation using the final application*

Summary

In this chapter, I showed you how to use the Web API and Knockout to create a single-page application that performs data operations using RESTful Web service. While not part of the MVC Framework, the Web API is modeled so closely on the nature and structure of MVC that it is familiar to MVC developers and, as I demonstrated, Web API controllers can be added alongside regular MVC controllers in an application.

And that is all I have to teach you about the MVC Framework. I started by creating a simple application, and then took you on a comprehensive tour of the different components in the framework, showing you how they can be configured, customized, or replaced entirely. I wish you every success in your MVC Framework projects and I can only hope that you have enjoyed reading this book as much as I enjoyed writing it.