CHAPTER 7

SportsStore: A Real Application

In the previous chapters, I built quick and simple MVC applications. I described the MVC pattern, the essential C# features and the kinds of tools that good MVC developers require. Now it is time to put everything together and build 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. 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 MVC Framework development is like by creating as realistic an example as possible. I want to focus on the MVC Framework, of course, and so I have simplified the integration with external systems, such as the database, and omitted others entirely, such as payment processing.

You might find the going a little slow as I build up the levels of infrastructure I need. Certainly, you *would* get the initial functionality built more quickly with Web Forms, just by dragging and dropping controls bound directly to a database. But the initial investment in an MVC application pays dividends, resulting in maintainable, extensible, well-structured code with excellent support for unit testing.

UNIT TESTING

I have made quite a big deal about the ease of unit testing in MVC, and about my belief that unit testing is an important part of the development process. You will see this demonstrated throughout this part of the book because I have included details of unit tests and techniques as they relate to key MVC features.

I know this is not a universal opinion. If you do not want to unit test, that is fine with me. To that end, when I have something to say that is purely about testing, I put it in a sidebar like this one. If you are not interested in unit testing, you can skip right over these sections, and the SportsStore application will work just fine. You do not need to do any kind of unit testing to get the technology benefits of ASP.NET MVC, although, of course, support for testing is a key reason for adopting the MVC Framework.

Most of the MVC features I use for the SportsStore application have their own chapters later in the book. Rather than duplicate everything here, I tell you just enough to make sense for the example application and point you to the other chapter for in-depth information.

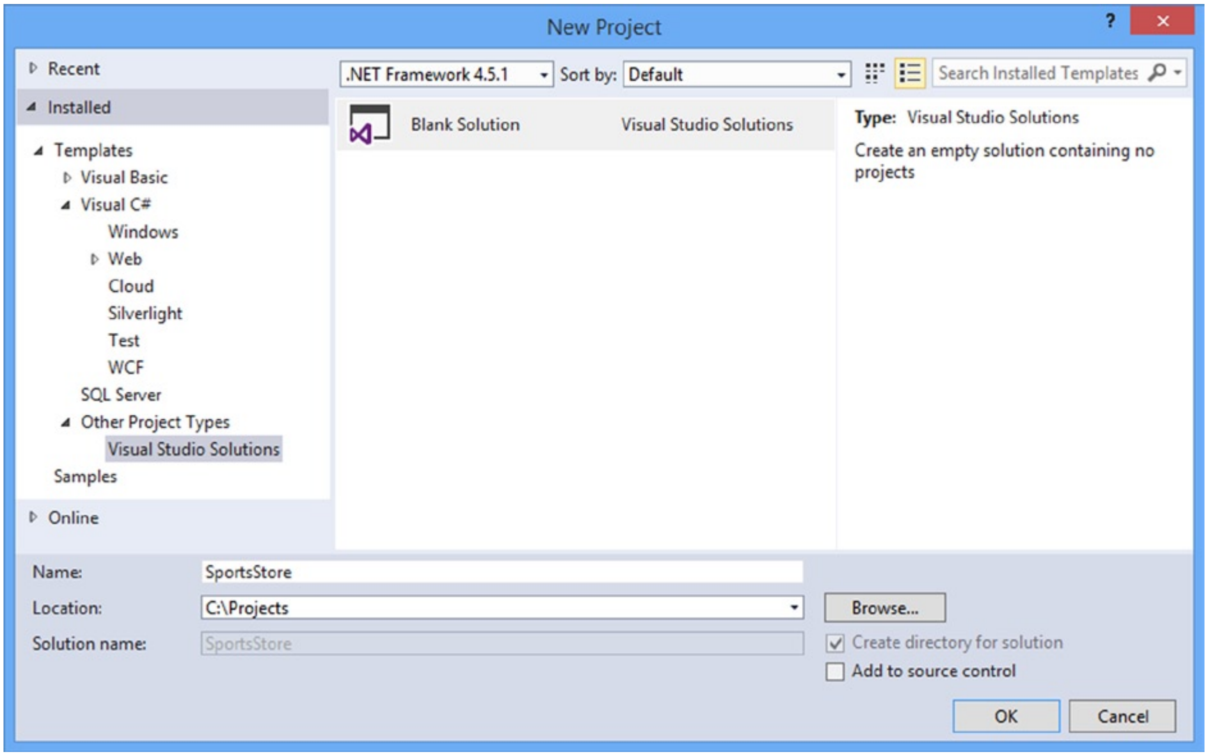
I will call out each step needed to build the application, so that you can see how the MVC features fit together. You should pay particular attention when I create views. You will get some odd results if you do not follow the examples closely.

Getting Started

You will need to install Visual Studio if you are planning to code the SportsStore application on your own computer as you read through this part of the book. You can also download the SportsStore project as part of the code archive that accompanies this book (available from `apress.com`). You do not need to follow along, of course. I have tried to make the screenshots and code listings as easy to follow as possible, just in case you are reading this book on a train, in a coffee shop, or the like.

Creating the Visual Studio Solution and Projects

I am going to create a Visual Studio solution that contains three projects. One project will contain the domain model, one will be the MVC application, and the third will contain the unit tests. To get started, I created a new Visual Studio solution called `SportsStore` using the `Blank Solution` template, which you can find in the `Other Project Types/Visual Studio Solutions` section of the `New Project` dialog, as shown in Figure 7-1. Click the `OK` button to create the solution.



Type: Visual Studio Solutions
Create an empty solution containing no projects

Name: SportsStore
Location: C:\Projects
Solution name: SportsStore
[✓] Create directory for solution
[ ] Add to source control

Figure 7-1. *Creating a new Visual Studio solution*

A Visual Studio solution is a container for one or more projects. I require three projects for my example app, which I have described in Table 7-1. You add a project by right-clicking the `Solution` entry in the Solution Explorer and selecting `Add` ➤ `New Project` from the pop-up menus.

Table 7-1. *The Three SportsStore Projects*

Project Name	Visual Studio Project Template	Purpose
`SportsStore.Domain`	Class Library	Holds the domain entities and logic; set up for persistence via a repository created with the Entity Framework.
`SportsStore.WebUI`	ASP.NET MVC Web Application (choose `Empty` when prompted to choose a project template and check the `MVC` option)	Holds the controllers and views; acts as the UI for the SportsStore application.
`SportsStore.UnitTests`	Unit Test Project	Holds the unit tests for the other two projects

I always use the `Empty` option for the `ASP.NET MVC Web Application` template. The other options add an initial setup to the project that includes JavaScript libraries, CSS style sheets, and C# classes that configure application features like security and routing. None of this is inherently bad–and some of the open-source libraries that Microsoft has recently "blessed" to be included in new projects are excellent–but you can manually set up all of the content and configuration and, in doing so, learn more about the workings of the MVC Framework.

When you have created the three projects, the Solution Explorer should look like Figure 7-2. I have deleted the `Class1.cs` file that Visual Studio adds to the `SportsStore.Domain` project. I will not be using it.

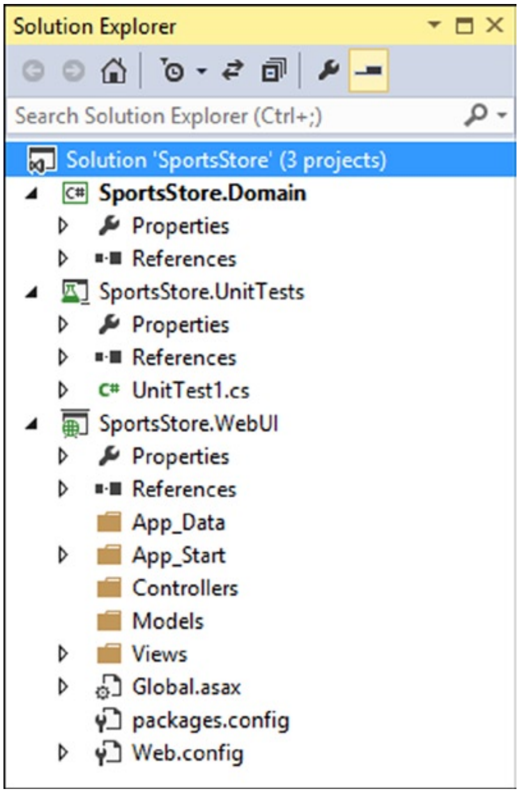


Figure 7-2. *The projects shown in the Solution Explorer window*

To make debugging easier, right-click the SportsStore.WebUI project and select Set as Startup Project from the pop-up menu (you will see the name turn bold). This means that when you select Start Debugging or Start without Debugging from the Debug menu, it is this project that will be started.

Visual Studio will try to navigate to individual view files if you are editing them when you start the debugger, so right-click the SportsStore.WebUI project in the Solution Explorer and select Properties from the pop-up menu. Click on Web to open the web-related properties and select the Specific Page option. There is no need to enter a value into the Specific Page text field. Just selecting the option is enough to stop Visual Studio from trying to guess the URL you want to view and ensure that the browser requests the root URL for the application when you start the debugger.

Installing the Tool Packages

I will be using Ninject and Moq in this chapter. Select Tools ➤ Library Package Manger ➤ Package Manager Console in Visual Studio to open the NuGet command line and enter the following commands:

```
Install-Package Ninject -version 3.0.1.10 -projectname SportsStore.WebUI
Install-Package Ninject.Web.Common -version 3.0.0.7 -projectname SportsStore.WebUI
Install-Package Ninject.MVC3 -Version 3.0.0.6 -projectname SportsStore.WebUI
Install-Package Ninject -version 3.0.1.10 -projectname SportsStore.UnitTests
Install-Package Ninject.Web.Common -version 3.0.0.7 -projectname SportsStore.UnitTests
Install-Package Ninject.MVC3 -Version 3.0.0.6 -projectname SportsStore.UnitTests
Install-Package Moq -version 4.1.1309.1617 -projectname SportsStore.WebUI
Install-Package Moq -version 4.1.1309.1617 -projectname SportsStore.UnitTests
Install-Package Microsoft.Aspnet.Mvc -version 5.0.0 -projectname SportsStore.Domain
Install-Package Microsoft.Aspnet.Mvc -version 5.0.0 -projectname SportsStore.UnitTests
```

There are many NuGet commands to enter because I am being selective about which packages NuGet installs into which projects and, as in previous chapters, I am specifying particular versions of the packages to download and install.

Adding References Between Projects

I need to set up dependencies between projects and to some of the Microsoft assemblies. Right-click each project in the Solution Explorer window, select Add Reference, and add the references shown in Table 7-2 from the Assemblies ➤ Framework, Assemblies ➤ Extensions or Solution sections.

Table 7-2. *Required Project Dependencies*

Project Name	Solution Dependencies	Assemblies References
SportsStore.Domain	None	System.ComponentModel.DataAnnotations
SportsStore.WebUI	SportsStore.Domain	None
SportsStore.UnitTests	SportsStore.Domain	System.Web
	SportsStore.WebUI	Microsoft.CSharp

■ **Caution** Take the time to set these relationships up properly. If you do not have the right libraries and project references, you will get into trouble when trying to build the project.

Setting Up the DI Container

In Chapter 6, I showed you how to use Ninject to create a custom dependency resolver that the MVC Framework will use to instantiate objects across the application. I am going to repeat that process, starting with adding an `Infrastructure` folder to the `SportsStore.WebUI` project and adding a class file called `NinjectDependencyResolver.cs` to it. You can see the contents of the new file in Listing 7-1.

Listing 7-1. The Contents of the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using Ninject;

namespace SportsStore.WebUI.Infrastructure {

    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
            // put bindings here
        }
    }
}
```

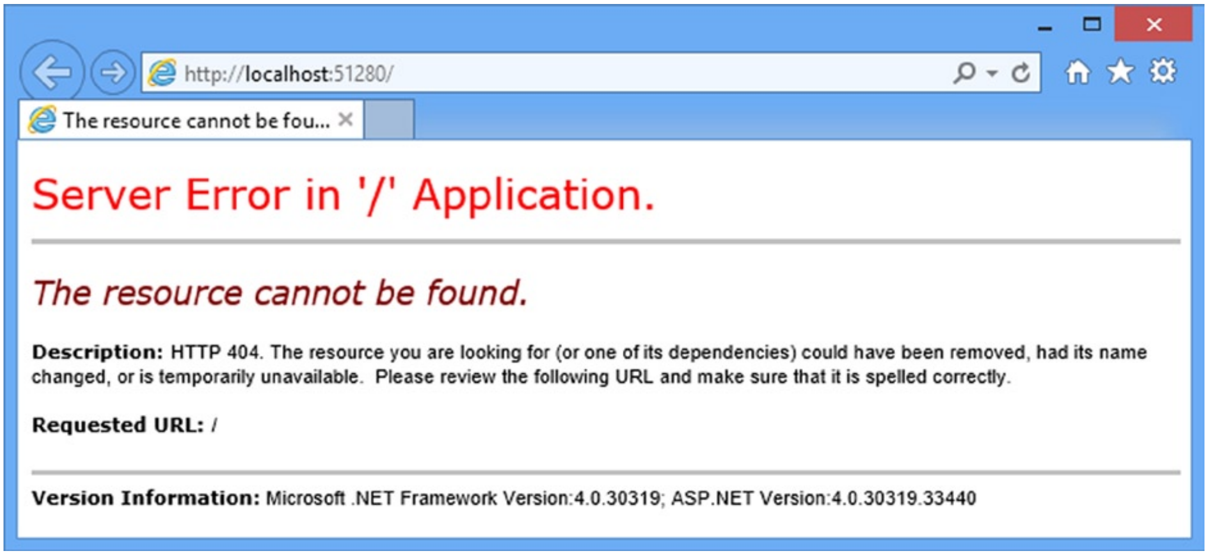
As you may recall from Chapter 6, the next step is to create a bridge between the `NinjectDependencyResolver` class and the MVC support for dependency injection in the `App_Start/NinjectWebCommon.cs` file, which one of the Ninject NuGet packages added to the project, as shown in Listing 7-2.

Listing 7-2. Integrating Ninject in the NinjectWebCommon.cs File

```
...
private static void RegisterServices(IKernel kernel) {
    System.Web.Mvc.DependencyResolver.SetResolver(new
        SportsStore.WebUI.Infrastructure.NinjectDependencyResolver(kernel));
}
...
```

Running the Application

If you select `Start Debugging` from the `Debug` menu, you will see an error page as shown in Figure 7-3. This is because you have requested a URL associated with a non-existent controller.



Server Error in '/' Application.

The resource cannot be found.

Description: HTTP 404. The resource you are looking for (or one of its dependencies) could have been removed, had its name changed, or is temporarily unavailable. Please review the following URL and make sure that it is spelled correctly.

Requested URL: /

Version Information: Microsoft .NET Framework Version:4.0.30319; ASP.NET Version:4.0.30319.33440

Figure 7-3. *The error page*

Starting the Domain Model

All MVC Framework projects start with the domain model because everything in an MVC Framework application revolves around it. Since this is an e-commerce application, the most obvious domain entity I need is a product. Create a new folder called `Entities` inside the `SportsStore.Domain` project and then a new C# class file called `Product.cs` within it. You can see the structure in Figure 7-4.

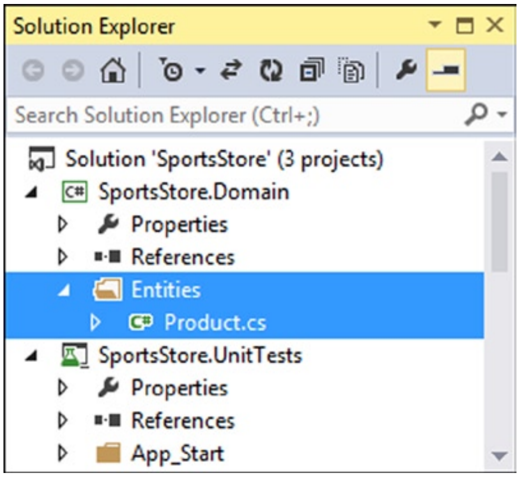


Figure 7-4. *Creating the Product class*

You are already familiar with the definition of the `Product` class, as I am going to use the one you saw in the previous chapters. Edit the `Product.cs` class file so that it matches Listing 7-3.

Listing 7-3. The Contents of the Product.cs File

```
namespace SportsStore.Domain.Entities {

    public class Product {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
    }
}
```

I am following the technique of defining my domain model in a separate Visual Studio project, which means that the class must be marked as `public`. You do not need to follow this convention, but I find that it helps keep the model separate from the controllers, which is useful in large and complex projects.

Creating an Abstract Repository

I need some way of getting `Product` entities from a database. As I explained in Chapter 3, the model includes the persistence logic for storing and retrieving the data from the persistent data store, but even within the model, I want to keep a degree of separation between the data model entities and the storage and retrieval logic, which I achieve using the *repository pattern*. I will not worry about how I am going to implement data persistence for the moment, but I *will* start the process of defining an interface for it.

Create a new top-level folder inside the `SportsStore.Domain` project called `Abstract` and, within the new folder, a new interface file called `IProductsRepository.cs`, the contents of which Listing 7-4 shows. You can add a new interface by right-clicking the `Abstract` folder, selecting `Add` ➤ `New Item`, and selecting the `Interface` template.

Listing 7-4. The Contents of the IProductRepository.cs File

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Abstract {
    public interface IProductRepository {

        IEnumerable<Product> Products { get; }
    }
}
```

This interface uses `IEnumerable<T>` to allow a caller to obtain a sequence of `Product` objects, without saying how or where the data is stored or retrieved. A class that depends on the `IProductRepository` interface can obtain `Product` objects without needing to know anything about where they are coming from or how the implementation class will deliver them. This is the essence of the repository pattern. I will revisit the `IProductRepository` interface throughout the development process to add features.

Making a Mock Repository

Now that I have defined an abstract interface, I could implement the persistence mechanism and hook it up to a database, but I want to add some of the other parts of the application first. In order to do this, I am going to create a mock implementation of the `IProductRepository` interface that will stand in until I return to the topic of data storage.

I define the mock implementation and bind it to the `IProductRepository` interface in the `AddBindings` method of the `NinjectDependencyResolver` class in the `SportsStore.WebUI` project, as illustrated by Listing 7-5.

Listing 7-5. Adding the Mock IProductRepository Implementation in the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using Moq;
using Ninject;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Infrastructure {

    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }
        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            mock.Setup(m => m.Products).Returns(new List<Product> {
                new Product { Name = "Football", Price = 25 },
                new Product { Name = "Surf board", Price = 179 },
                new Product { Name = "Running shoes", Price = 95 }
            });

            kernel.Bind<IProductRepository>().ToConstant(mock.Object);
        }
    }
}
```

I had to add a number of namespaces to the file for this addition, but the process I used to create the mock repository implementation uses the same Moq techniques I introduced in Chapter 6. I want Ninject to return the same mock object whenever it gets a request for an implementation of the `IProductRepository` interface, which is why I used the `ToConstant` method to set the Ninject scope, like this:

```
...
kernel.Bind<IProductRepository>().ToConstant(mock.Object);
...
```

Rather than create a new instance of the implementation object each time, Ninject will always satisfy requests for the `IProductRepository` interface with the same mock object.

Displaying a List of Products

I could spend the rest of this chapter building out the domain model and the repository, and not touch the UI project at all. I think you would find that boring, though, so I am going to switch tracks and start using the MVC Framework in earnest. I will add model and repository features as I need them.

In this section, I am going to create a controller and an action method that can display details of the products in the repository. For the moment, this will be for only the data in the mock repository, but I will sort that out later. I will also set up an initial *routing configuration* so that MVC knows how to map requests for the application to the controller I create.

Adding a Controller

Right-click the `Controllers` folder in the `SportsStore.WebUI` project and select `Add` ➤ `Controller` from the pop-up menu. Select the `MVC 5 Controller - Empty` option, click the `Add` button and set the name to `ProductController`. Click the `Add` button and Visual Studio will create a new class file called `ProductController.cs`, which you should edit to match Listing 7-6.

Listing 7-6. The Initial Contents of the Product Controller.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;

        public ProductController(IProductRepository productRepository) {
            this.repository = productRepository;
        }
    }
}
```

In addition to removing the `Index` action method, I added a constructor that declares a dependency on the `IProductRepository` interface, which will lead Ninject to inject the dependency for the product repository when it instantiates the controller class. I also imported the `SportsStore.Domain` namespaces so that I can refer to the repository and model classes without having to qualify their names. Next, I have added an action method, called `List`, which will render a view showing the complete list of products, as shown in Listing 7-7.

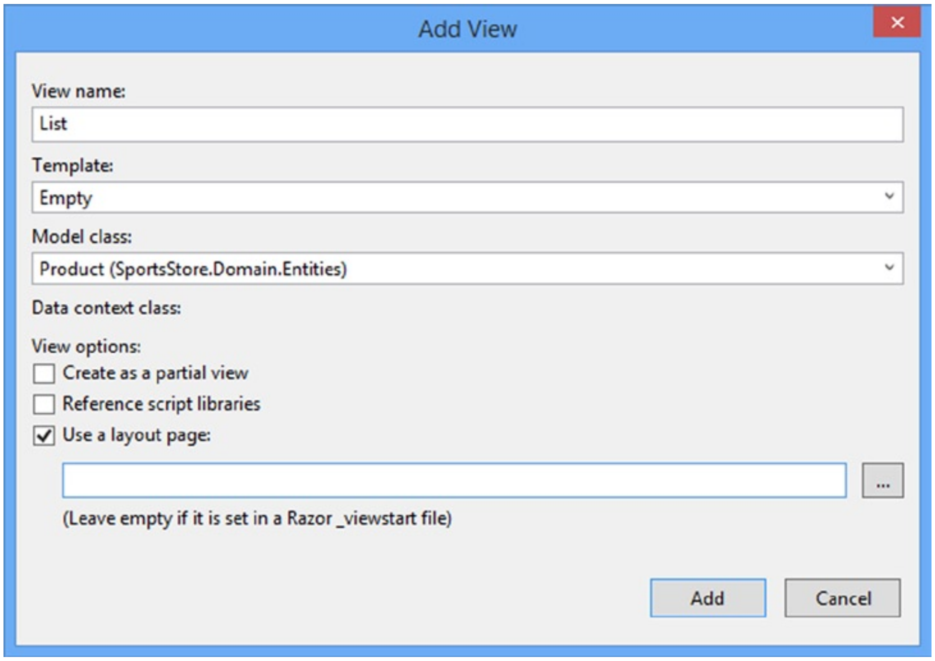
Listing 7-7. Adding an Action Method to the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;

        public ProductController(IProductRepository productRepository) {
            this.repository = productRepository;
        }

        public ViewResult List() {
            return View(repository.Products);
        }
    }
}
```

Calling the `View` method like this (without specifying a view name) tells the framework to render the default view for the action method. Passing a `List` of `Product` objects to the `View` method, provides the framework with the data with which to populate the `Model` object in a strongly typed view.

Adding the Layout, View Start File and View

Now I need to add the default view for the `List` action method. Right-click on the `List` action method in the `HomeController` class and select `Add View` from the pop-up menu. Set `View Name` to `List`, set `Template` to `Empty`, and select `Product` for the `Model Class`, as shown in Figure 7-5. Ensure that the `Use A Layout Page` box is checked and click the `Add` button to create the view.



View name: List

Template: Empty

Model class: Product (SportsStore.Domain.Entities)

Data context class:

View options:
- [ ] Create as a partial view
- [ ] Reference script libraries
- [✓] Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

Figure 7-5. *Adding the Views/Product/List.cshtml view*

When you click the `Add` button, Visual Studio will create the `List.cshtml` file, but it will also create a `_ViewStart.cshtml` file and a `Shared/_Layout.cshtml` file. This is a helpful feature, but in keeping with the Microsoft approach to default content, the `_Layout.cshtml` file contains template content that I do not want or need. Edit the layout so that it matches the content shown in Listing 7-8.

Listing 7-8. Editing the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title</title>
</head>
```

```
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Rendering the View Data

Although I set the model type of the view to be the `Product` class, I actually want to work with an `IEnumerable<Product>`, which is what the `Product` controller obtains from the repository and passes to the view. In Listing 7-9, you can see that I have edited the `@model expression` and added some HTML and Razor expressions to display details of the products.

Listing 7-9. Editing the List.cshtml File

```
@using SportsStore.Domain.Entities
@model IEnumerable<Product>

@{
    ViewBag.Title = "Products";
}


@foreach (var p in Model) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
```

I also changed the title of the page. Notice that I do not need to use the Razor `@:` expression to display the view data. This is because each of the content lines in the code body either is a Razor directive or starts with an HTML element.

■ **Tip** I converted the `Price` property to a string using the `ToString("c")` method, which renders numerical values as currency according to the culture settings that are in effect on your server. For example, if the server is set up as `en-US`, then `(1002.3).ToString("c")` will return `$1,002.30`, but if the server is set to `en-GB`, then the same method will return `£1,002.30`. You can change the culture setting for your server by adding a section to the `<system.web>` node in the `Web.config` file like this: `<globalization culture="en-GB" uiCulture="en-GB" />`.

Setting the Default Route

I need to tell the MVC Framework that it should send requests that arrive for the root URL of my application (`http://mysite/`) to the `List` action method in the `ProductController` class. I do this by editing the statement in the `RegisterRoutes` method in the `App_Start/RouteConfig.cs` file, as shown in Listing 7-10.

Listing 7-10. Adding the Default Route in the RouteConfig.cs File

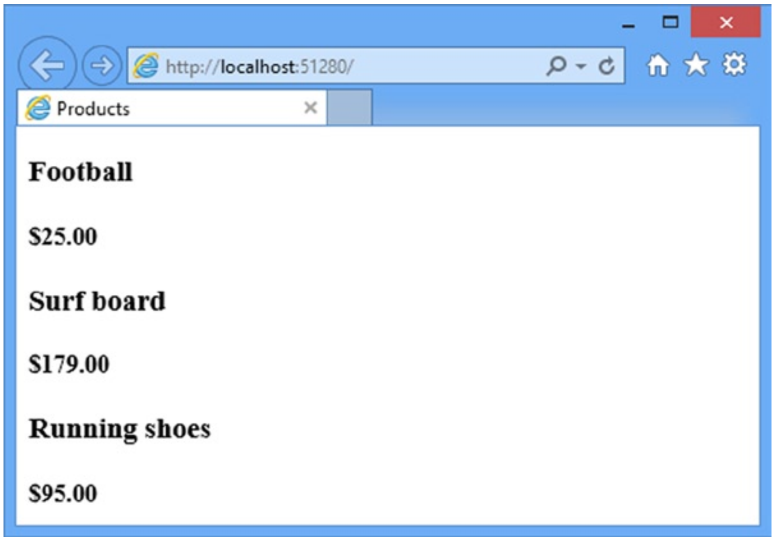
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace SportsStore.WebUI {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Product", action = "List",
                    id = UrlParameter.Optional }
            );
        }
    }
}
```

You can see the changes in bold. Change Home to Product and Index to List, as shown in the listing. I cover the ASP.NET routing feature in detail in Chapters 15 and 16. For now, it is enough to know that this change directs requests for the default URL to the List action method in the Product controller.

■ **Tip** Notice that I have set the value of the controller in Listing 7-10 to be Product and not ProductController, which is the name of the class. This is part of the ASP.NET MVC naming scheme, in which controller classes *always* end in Controller but you omit this part of the name when referring to the class.

Running the Application

I have all the basics in place. I have a controller with an action method that the MVC Framework will call when the default URL is requested. That action method relies on a mock implementation of the repository interface, which generates some simple test data. The controller passes the test data to the view that I associated with the action method, and the view displays a simple list of the details for each product. You can see the result by running the application, as shown in Figure 7-6. If you don't get the result in the figure, check that you have navigated to the root URL and not one that targets another action.



Football

$25.00

Surf board

$179.00

Running shoes

$95.00

Figure 7-6. *Viewing the basic application functionality*

This is the typical pattern of development for the ASP.NET MVC Framework. An initial investment of time setting everything up is necessary, and then the basic features of the application snap together quickly.

EASIER DEBUGGING

When you run the project from the `Debug` menu, Visual Studio will create a new browser window to display the application, which can take a few seconds. There are some tricks that you can use to speed the process up.

If you are editing view files and not classes, then you can make changes in Visual Studio while the debugger is running. Reload the browser window when you want to see the effect of your changes. ASP.NET will recompile your views into classes and display the changes immediately. Visual Studio won't let you edit class files when the debugger is running or make some kinds of change to the project in the Solution Explorer, so this technique is most useful when you are tweaking the fit and finish of the HTML that your application generates.

Visual Studio 2013 includes a new feature called *browser link* that lets you have multiple browser windows open and to reload them from the Visual Studio menu bar. I demonstrate this feature in Chapter 14.

As a final alternative, you can keep your application open in a stand-alone browser window. To do this (assuming you have launched the debugger at least once already), right-click the IIS Express icon in the system tray and select the URL for your app from the pop-up menu. After you have made your changes, compile the solution in Visual Studio by pressing F6 or choosing `Build` ➤ `Build Solution`, and then switch to your browser window and reload the Web page.

Preparing a Database

I can already display simple views that contain details of the products, but I am displaying the test data that the mock `IproductRepository` returns. Before I can implement a real repository, I need to set up a database and populate it with some data.

I am going to use SQL Server as the database, and I will access the database using the Entity Framework (EF), which is the Microsoft.NET ORM framework. An ORM framework presents the tables, columns, and rows of a relational database through regular C# objects. I mentioned in Chapter 6 that LINQ can work with different sources of data, and one of these is the Entity Framework. You will see how this simplifies things in a little while.

■ **Note** This is an area where you can choose from a wide range of tools and technologies. Not only are there different relational databases available, but you can also work with object repositories, document stores, and some esoteric alternatives. There are other .NET ORM frameworks as well, each of which takes a slightly different approach: variations that may give you a better fit for your projects.

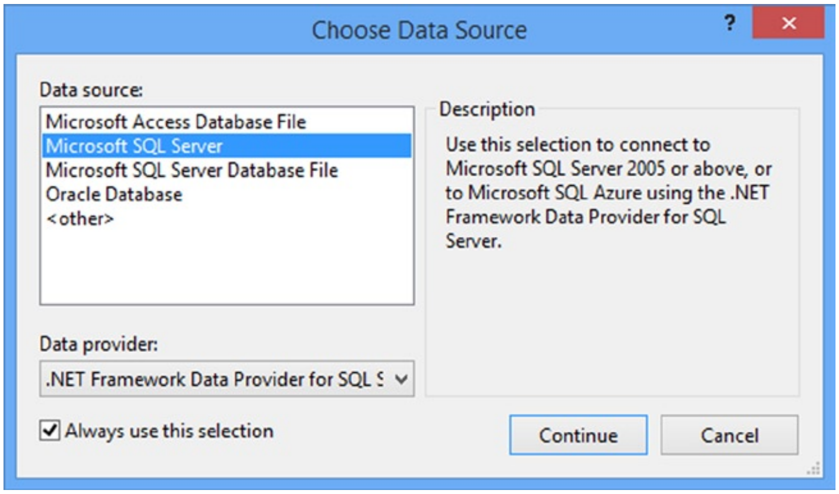
I am using the Entity Framework for a several reasons: it is simple and easy to get it up and working; the integration with LINQ is first rate (and I like using LINQ); and it is good. The earlier releases were a bit hit-and-miss, but the current versions are elegant and feature-rich.

Creating the Database

A nice feature of Visual Studio and SQL Server is the *LocalDB* feature, which is an administration-free implementation of the core SQL Server features specifically designed for developers. Using this feature, I can skip the process of setting up a database while I build my project and then deploy to a full SQL Server instance. Most MVC applications are deployed to hosted environments that are run by professional administrators and so the LocalDB feature means that database configuration can be left in the hands of DBAs and developers can get on with coding. The LocalDB feature is installed automatically with Visual Studio Express 2013 for Web, but you can download it directly from `www.microsoft.com/sqlserver` if you prefer.

The first step is to create the database connection in Visual Studio. Open the `Server Explorer` window from the `View` menu and click the `Connect to Database` button (it looks like a power cable with a green plus sign).

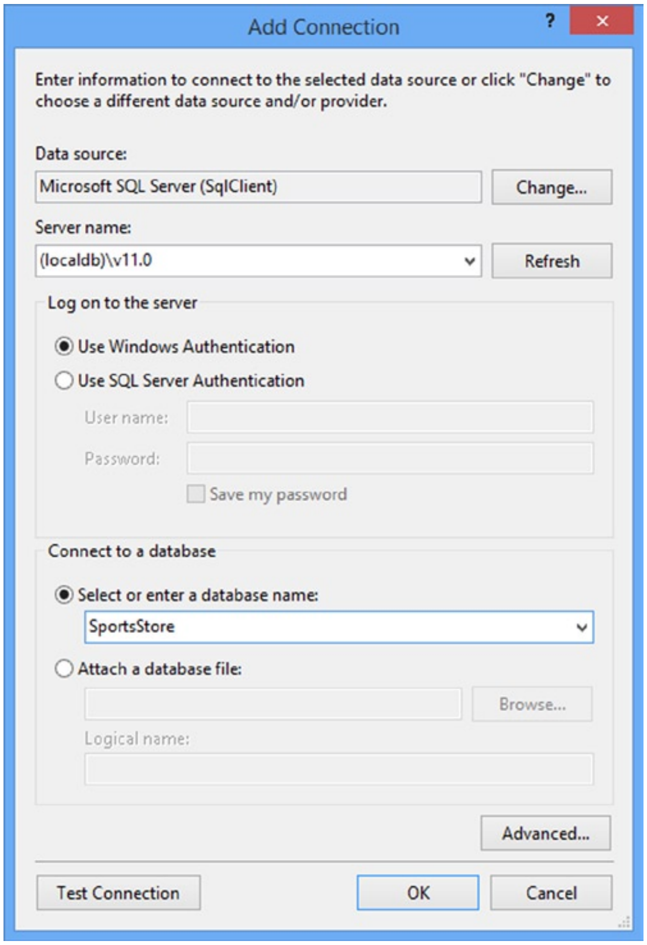
You will see the `Choose Data Source` dialog. Select the `Microsoft SQL Server` option, as shown in Figure 7-7, and click the `Continue` button. (Visual Studio remembers the selection you make, so you will not see this window if you already created a database connection in another project).


Choose Data Source
Data source:
Microsoft Access Database File
Microsoft SQL Server
Microsoft SQL Server Database File
Oracle Database
<other>
Description
Data provider:
.NET Framework Data Provider for SQL S
Always use this selection
Continue
Cancel

Use this selection to connect to Microsoft SQL Server 2005 or above, or to Microsoft SQL Azure using the .NET Framework Data Provider for SQL Server.

Figure 7-7. Selecting the Data Source

Next, you will see the `Add Connection` dialog. Set the server name to `(localdb)\v11.0`. This is a special name that indicates that you want to use the LocalDB feature. Check the `Use Windows Authentication` option and set the database name to `SportsStore`, as shown by Figure 7-8.


Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server (SqlClient)

Server name: (localdb)\v11.0

Log on to the server

[✓ Use Windows Authentication]

Use SQL Server Authentication

User name:

Password:

Save my password

Connect to a database

[✓ Select or enter a database name:] SportsStore

Attach a database file:

Logical name:

***Figure 7-8.** Setting up the SportsStore database*

■ **Tip** If you did not see the Choose Data Source dialog, you can click the `Change` button in the top right of the Add Connection dialog.

Click the `OK` button and Visual Studio will prompt you to create the new database: click the `Yes` to go ahead. A new item will appear in the `Data Connections` section of the Server Explorer window, which you can expand to see the different facets of the database, as shown in Figure 7-9. You should see something similar, but the name of the database connection will be different because it will include the local PC name (the name of my workstation is `tiny`).

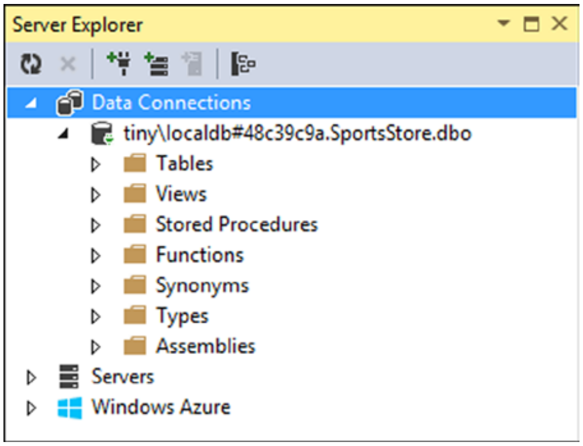


Figure 7-9. *The LocalDB database as shown in the Server Explorer window*

Defining the Database Schema

As I explained at the start of the chapter, my focus with the SportsStore application is to focus on the MVC Framework development process, and that means keeping the other components that the application relies on as simple as possible. I do not want to get into the topics of database design and the in-depth details of the Entity Framework, beyond what I need to demonstrate how to get data in and out of an application. These are topics in their own right and they are not part of ASP.NET or the MVC Framework.

With this in mind, I am going to use a database that contains only one table. This is not how real e-commerce sites would structure their data, of course, but the important lesson in this section is about the repository pattern and how I use it to store and retrieve data, not the structure of the database.

To create the database table, right-click the `Tables` item for the new SportsStore database in the Server Explorer window and select `Add New Table`, as shown in Figure 7-10.

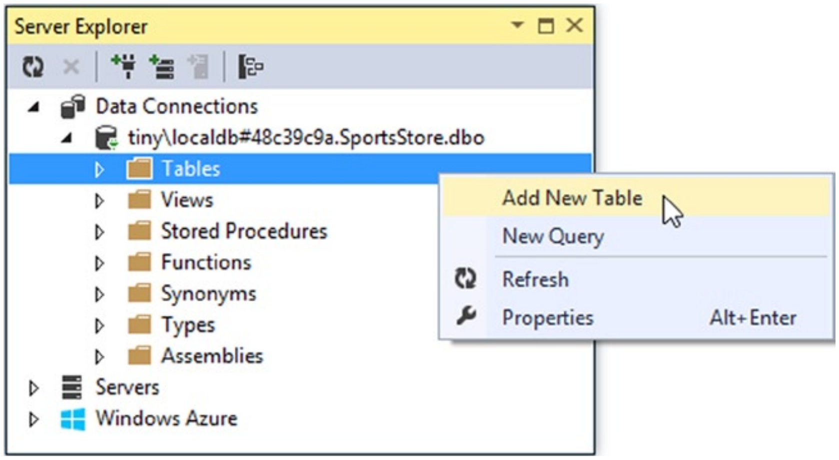


Figure 7-10. *Adding a new table*

Visual Studio will display a designer window for creating a new table. You can create new database tables using the visual part of the designer, but I am going to use the T-SQL section because it is a more concise and accurate way of describing the table specification I require in a book. Enter the SQL statement shown in Listing 7-11 and click the `Update` button in the top-left corner of the table design window.

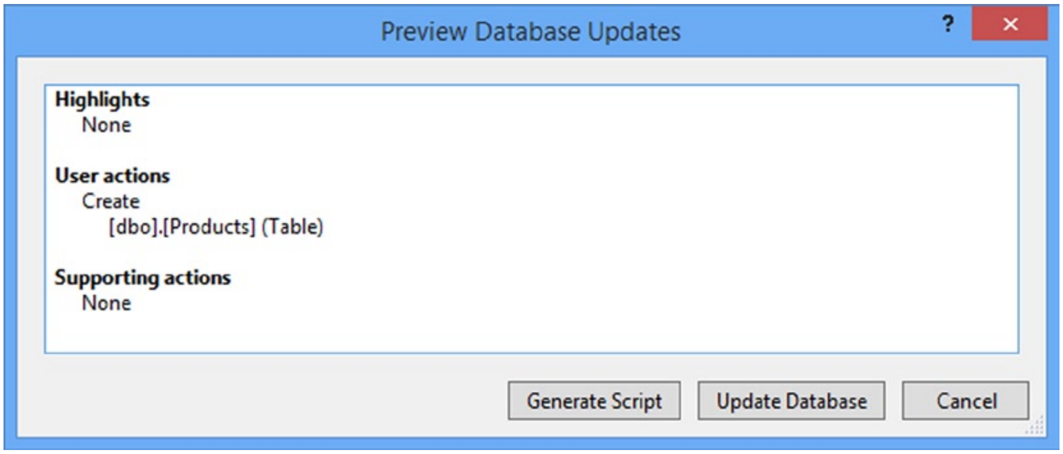
Listing 7-11. The SQL Statement to Create the Table in the SportsStore Database

```
CREATE TABLE Products
(
    [ProductID] INT NOT NULL PRIMARY KEY IDENTITY,
    [Name] NVARCHAR(100) NOT NULL,
    [Description] NVARCHAR(500) NOT NULL,
    [Category] NVARCHAR(50) NOT NULL,
    [Price] DECIMAL(16, 2) NOT NULL
)
```

This statement creates a table called `Products`, which has columns for the different properties I defined in the `Product` model class earlier in the chapter.

■ **Tip** Setting the `IDENTITY` property for the `ProductID` column means that SQL Server will generate a unique primary key value when I add data to this table. When using a database in a Web application, it can be difficult to generate unique primary keys because requests from users arrive concurrently. By enabling this feature, I can store new table rows and rely on SQL Server to sort out unique values.

When you click the `Update` button, Visual Studio will show a summary of the effect of the statement, as shown in Figure 7-11.



Highlights
None

User actions
Create
[dbo].[Products] (Table)

Supporting actions
None

Figure 7-11. *The summary of the effect of the SQL statement*

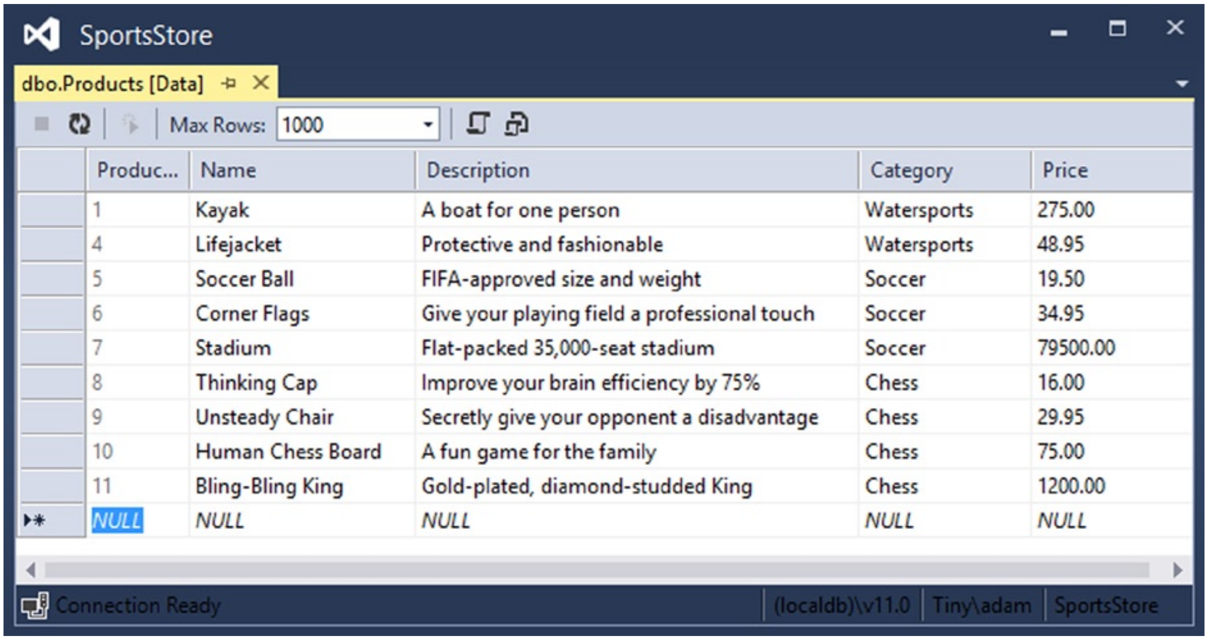
Click the `Update Database` button to execute the SQL and create the `Products` table in the database. You will be able to see the effect the update has if you click the `Refresh` button in the Server Explorer window. The `Tables` section shows the new `Product` table and details of each of the rows.

■ **Tip** After you have updated the database, you can close the `dbo.Products` window. Visual Studio will offer you the chance to save the SQL script used to create the database. You do not need to save the script for this chapter, but it can be useful in real projects if you need to configure multiple databases.

Adding Data to the Database

I am going to add data to the database so that I have something to work with until I add the catalog administration features in Chapter 11.

In the `Server Explorer` window, expand the `Tables` item of the `SportsStore` database, right-click the `Products` table, and select `Show Table Data`. Enter the data shown in Figure 7-12. You can move from row to row by using the Tab key. At the end of each row, pressing tab will move to the next row and update the data in the database.



	Produc...	Name	Description	Category	Price
	1	Kayak	A boat for one person	Watersports	275.00
	4	Lifejacket	Protective and fashionable	Watersports	48.95
	5	Soccer Ball	FIFA-approved size and weight	Soccer	19.50
	6	Corner Flags	Give your playing field a professional touch	Soccer	34.95
	7	Stadium	Flat-packed 35,000-seat stadium	Soccer	79500.00
	8	Thinking Cap	Improve your brain efficiency by 75%	Chess	16.00
	9	Unsteady Chair	Secretly give your opponent a disadvantage	Chess	29.95
	10	Human Chess Board	A fun game for the family	Chess	75.00
	11	Bling-Bling King	Gold-plated, diamond-studded King	Chess	1200.00
▸*	NULL	NULL	NULL	NULL	NULL

***Figure 7-12.** Adding data to the Products table*

■ **Note** You must leave the `ProductID` column empty. It is an identity column, so SQL Server will generate a unique value when you tab to the next row.

I listed the product details in Table 7-3 in case you cannot make out the detail from the figure. It doesn't matter if you don't enter all of the details exactly as I have, although you'll see different results from the ones I show as you work through the process of creating the rest of the SportsStore application.

Table 7-3. *The Data for the Products Table*

Name	Description	Category	Price
Kayak	A boat for one person	Watersports	275.00
Lifejacket	Protective and fashionable	Watersports	48.95
Soccer Ball	FIFA-approved size and weight	Soccer	19.50
Corner Flags	Give your playing field a professional touch	Soccer	34.95
Stadium	Flat-packed, 35,000-seat stadium	Soccer	79,500.00
Thinking Cap	Improve your brain efficiency by 75%	Chess	16.00
Unsteady Chair	Secretly give your opponent a disadvantage	Chess	29.95
Human Chess Board	A fun game for the family	Chess	75.00
Bling-Bling King	Gold-plated, diamond-studded King	Chess	1,200.00

Creating the Entity Framework Context

Recent versions of the Entity Framework include a nice feature called *code-first*. The idea is that I can define classes in my model and then generate a database from those classes.

This is great for green-field development projects, but these are few and far between. Instead, I am going to show you a variation on code-first, where I associate the model classes with an existing database. Select `Tools` ➤ `Library Package Manager` ➤ `Package Manager Console` in Visual Studio to open the NuGet command line and enter the following command:

```
Install-Package EntityFramework -projectname SportsStore.Domain
Install-Package EntityFramework -projectname SportsStore.WebUI
```

■ **Tip** You may see errors in the `Package Manager Console` telling you that *binding redirects* cannot be created. You can safely ignore these warnings.

This command adds the Entity Framework package to the solution. I need to install the same package in the `Domain` and `WebUI` projects so that I create the classes that will access the database in the `Domain` project and access the database in the `WebUI` project.

The next step is to create a *context* class that will associate the model with the database. Create a new folder in the `SportsStore.Domain` project called `Concrete` and add a new class file called `EFDbContext.cs` within it. Edit the contents of the class file so they match Listing 7-12.

Listing 7-12. The Content of the EFDbContext.cs File

```
using SportsStore.Domain.Entities;
using System.Data.Entity;

namespace SportsStore.Domain.Concrete {
    public class EFDbContext : DbContext {
        public DbSet<Product> Products { get; set; }
    }
}
```

To take advantage of the code-first feature, I need to create a class that is derived from System.Data.Entity.DbContext. This class then automatically defines a property for each table in the database that I want to work with.

The name of the property specifies the table, and the type parameter of the DbSet result specifies the model type that the Entity Framework should use to represent rows in that table. In this case, the property name is Products and the type parameter is Product, meaning that the Entity Framework should use the Product model type to represent rows in the Products table.

Next, I need to tell the Entity Framework how to connect to the database, which I do by adding a database connection string to the Web.config file in the SportsStore.WebUI project with the same name as the context class, as shown in Listing 7-13.

Listing 7-13. Adding a Database Connection in the Web.config File

```
<?xml version="1.0" encoding="utf-8"?>

<configuration>

  <connectionStrings>
    <add name="EFDbContext" connectionString="Data Source=(localdb)\v11.0;Initial
        Catalog=SportsStore;Integrated Security=True"
        providerName="System.Data.SqlClient"/>
  </connectionStrings>

  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
     <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5.1" />
    <httpRuntime targetFramework="4.5.1" />
  </system.web>
</configuration>
```

■ **Tip** Notice that I have switched project here. I define the model and the repository logic in the SportsStore.Domain project, but the database connection information is put in the Web.config file in the SportsStore.WebUI project.

■ **Caution** I have had to split the value of the connectionString attribute across multiple lines to fit it on the page, but it is important to put everything on a single line in the Web.config file.

There will be another add element in the connectionsStrings section of the Web.config file. Visual Studio creates this element by default and you can either ignore it or, as I have, delete it from the Web.config file.

Creating the Product Repository

All that remains is to add a class file to the Concrete folder of the SportsStore.Domain project called EFProductRepository.cs. Edit your class file so it matches Listing 7-14.

Listing 7-14. The Contents of the EFProductRepostory.cs File

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System.Collections.Generic;

namespace SportsStore.Domain.Concrete {

    public class EFProductRepository : IProductRepository {
        private EFDbContext context = new EFDbContext();

        public IEnumerable<Product> Products {
            get { return context.Products; }
        }
    }
}
```

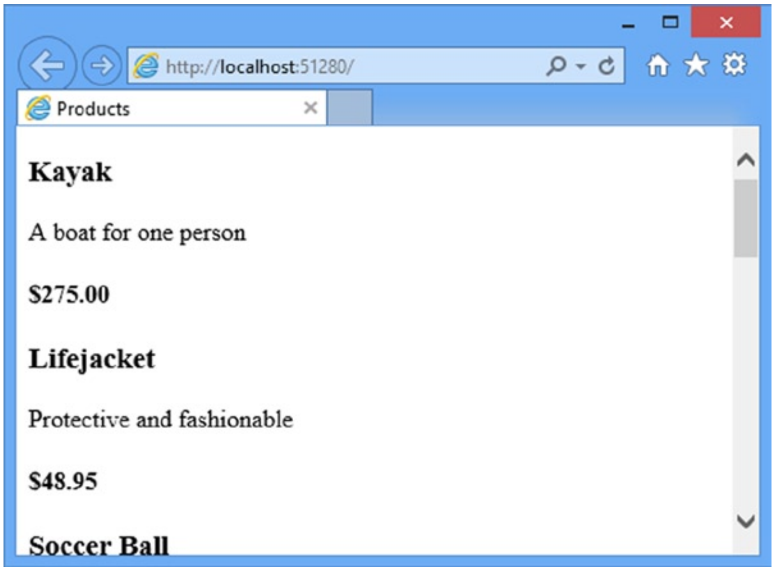
This is the repository class. It implements the IProductRepository interface and uses an instance of EFDbContext to retrieve data from the database using the Entity Framework. You will see how I work with the Entity Framework (and how simple it is) as I add features to the repository.

To use the new repository class, I need to edit the Ninject bindings and replace the mock repository with a binding for the real one. Edit the NinjectDependencyResolver.cs class file in the SportsStore.WebUI project so that the AddBindings method looks like Listing 7-15.

Listing 7-15. Adding the Real Repository Binding in the NinjectDependencyResolver.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using Moq;
using Ninject;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Concrete;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Infrastructure {

    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver(IKernel kernelParam) {
            kernel = kernelParam;
            AddBindings();
        }
        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        private void AddBindings() {
            kernel.Bind<IProductRepository>().To<EFProductRepository>();
        }
    }
}
```

The new binding is in bold. It tells Ninject to create instances of the `EFProductRepository` class to service requests for the `IProductRepository` interface. All that remains now is to run the application again. Figure 7-13 shows the results, which demonstrate the application is getting its product data from the database, rather than the mock repository.



Kayak

A boat for one person

$275.00

Lifejacket

Protective and fashionable

$48.95

Soccer Ball

Figure 7-13. The result of implementing the real repository

■ **Tip** If you get a `System.ArgumentException` when you start the project, then you have split the details of the database connection over two lines in the `Web.config` file. See the previous section for details.

This approach to getting the Entity Framework to present a SQL Server database as a series of model objects is simple and easy to work with, and it allows me to keep my focus on the MVC Framework. Of course, I am skipping over a lot of the detail in how the Entity Framework operates and the huge number of configuration options that are available. I like the Entity Framework a lot and I recommend that you spend some time getting to know it in detail. A good place to start is the Microsoft site for the Entity Framework: http://msdn.microsoft.com/data/ef.

Adding Pagination

You can see from Figure 7-13 that the `List.cshtml` view displays all of the products in the database on a single page. In this section, I will add support for pagination so that the view displays a number of products on a page, and the user can move from page to page to view the overall catalog. To do this, I am going to add a parameter to the `List` method in the `Product` controller, as shown in Listing 7-16.

Listing 7-16. Adding Pagination Support to the List Action Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository productRepository) {
            this.repository = productRepository;
        }

        public ViewResult List(int page = 1) {
            return View(repository.Products
                .OrderBy(p => p.ProductID)
                .Skip((page - 1) * PageSize)
                .Take(PageSize));
        }
    }
}
```

The `PageSize` field specifies that I want four products per page. I will come back and replace this with a better mechanism later on. I have added an *optional parameter* to the `List` method. This means that if I call the method without a parameter (`List()`), my call is treated as though I had supplied the value specified in the parameter definition (`List(1)`). The effect is that the action method displays the first page of products when the MVC Framework invokes it without an argument. Within the body of the action method, I get the `Product` objects, order them by the primary key, skip over the products that occur before the start of the current page, and take the number of products specified by the `PageSize` field.

UNIT TEST: PAGINATION

I can unit test the pagination feature by creating a mock repository, injecting it into the constructor of the `ProductController` class, and then calling the `List` method to request a specific page. I can then compare the `Product` objects I get with what I would expect from the test data in the mock implementation. See Chapter 6 for details of how to set up unit tests. Here is the unit test I created for this purpose, in the `UnitTest1.cs` file of the `SportsStore.UnitTests` project:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;

namespace SportsStore.UnitTests {

    [TestClass]
    public class UnitTest1 {

        [TestMethod]
        public void Can_Paginate() {

            // Arrange
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            mock.Setup(m => m.Products).Returns(new Product[] {
                new Product {ProductID = 1, Name = "P1"},
                new Product {ProductID = 2, Name = "P2"},
                new Product {ProductID = 3, Name = "P3"},
                new Product {ProductID = 4, Name = "P4"},
                new Product {ProductID = 5, Name = "P5"}
            });

            ProductController controller = new ProductController(mock.Object);
            controller.PageSize = 3;

            // Act
            IEnumerable<Product> result =
                (IEnumerable<Product>)controller.List(2).Model;

            // Assert
            Product[] prodArray = result.ToArray();
            Assert.IsTrue(prodArray.Length == 2);
            Assert.AreEqual(prodArray[0].Name, "P4");
            Assert.AreEqual(prodArray[1].Name, "P5");
        }
    }
}
```

Notice how easy it is to get the data that returned from a controller method. I call the `Model` property on the result to get the `IEnumerable<Product>` sequence generated in the `List` method. I then check that the data is what I expect. In this case, I converted the sequence to an array using the LINQ `ToArray` extension method and checked the length and the values of the individual objects.

Displaying Page Links

If you run the application, you will see that there are only four items shown on the page. If you want to view another page, you can append query string parameters to the end of the URL, like this:

```
http://localhost:51280/?page=2
```

You will need to change the port part of the URL to match whatever port your ASP.NET development server is running on. Using these query strings, you can navigate through the catalog of products.

Of course, there is no way for customers to figure out that these query string parameters exist, and even if there were, they are not going to want to navigate this way. Instead, I need to render some page links at the bottom of the each list of products so that customers can navigate between pages. To do this, I am going to implement a reusable HTML helper method, similar to the `Html.TextBoxFor` and `Html.BeginForm` methods I used in Chapter 2. The helper will generate the HTML markup for the navigation links I require.

Adding the View Model

To support the HTML helper, I am going to pass information to the view about the number of pages available, the current page, and the total number of products in the repository. The easiest way to do this is to create a view model, which I introduced briefly in Chapter 3. Add the class shown in Listing 7-17, called `PagingInfo`, to the `Models` folder in the `SportsStore.WebUI` project.

Listing 7-17. The Contents of the PagingInfo.cs File

```
using System;

namespace SportsStore.WebUI.Models {

    public class PagingInfo {
        public int TotalItems { get; set; }
        public int ItemsPerPage { get; set; }
        public int CurrentPage { get; set; }

        public int TotalPages {
            get { return (int)Math.Ceiling((decimal)TotalItems / ItemsPerPage); }
        }
    }
}
```

A view model is not part of the domain model. It is just a convenient class for passing data between the controller and the view. To emphasize this, I put this class in the `SportsStore.WebUI` project to keep it separate from the domain model classes.

Adding the HTML Helper Method

Now that I have a view model, I can implement the HTML helper method, which I am going to call PageLinks. Create a new folder in the SportsStore.WebUI project called HtmlHelpers and add a new class file called PagingHelpers.cs, the contents of which Listing 7-18 shows.

Listing 7-18. The Contents of the PagingHelpers.cs Class File

```
using System;
using System.Text;
using System.Web.Mvc;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.HtmlHelpers {

    public static class PagingHelpers {

        public static MvcHtmlString PageLinks(this HtmlHelper html,
                                              PagingInfo pagingInfo,
                                              Func<int, string> pageUrl) {

            StringBuilder result = new StringBuilder();
            for (int i = 1; i <= pagingInfo.TotalPages; i++) {
                TagBuilder tag = new TagBuilder("a");
                tag.MergeAttribute("href", pageUrl(i));
                tag.InnerHtml = i.ToString();
                if (i == pagingInfo.CurrentPage) {
                    tag.AddCssClass("selected");
                    tag.AddCssClass("btn-primary");
                }
                tag.AddCssClass("btn btn-default");
                result.Append(tag.ToString());
            }
            return MvcHtmlString.Create(result.ToString());
        }
    }
}
```

The PageLinks extension method generates the HTML for a set of page links using the information provided in a PagingInfo object. The Func parameter accepts a delegate that it uses to generate the links to view other pages.

UNIT TEST: CREATING PAGE LINKS

To test the `PageLinks` helper method, I call the method with test data and compare the results to the expected HTML. The unit test method is as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;
using SportsStore.WebUI.Models;
using SportsStore.WebUI.HtmlHelpers;

namespace SportsStore.UnitTests {
    [TestClass]
    public class UnitTest1 {

        [TestMethod]
        public void Can_Paginate() {
            // ...statements removed for brevity...
        }

        [TestMethod]
        public void Can_Generate_Page_Links() {

            // Arrange - define an HTML helper - we need to do this
            // in order to apply the extension method
            HtmlHelper myHelper = null;

            // Arrange - create PagingInfo data
            PagingInfo pagingInfo = new PagingInfo {
                CurrentPage = 2,
                TotalItems = 28,
                ItemsPerPage = 10
            };

            // Arrange - set up the delegate using a lambda expression
            Func<int, string> pageUrlDelegate = i => "Page" + i;

            // Act
            MvcHtmlString result = myHelper.PageLinks(pagingInfo, pageUrlDelegate);
```

```
                // Assert
                Assert.AreEqual(@"<a class=""btn btn-default"" href=""Page1"">1</a>"
            + @"<a class=""btn btn-default btn-primary selected"" href=""Page2"">2</a>"
                    + @"<a class=""btn btn-default"" href=""Page3"">3</a>",
                    result.ToString());
            }
        }
    }
```

This test verifies the helper method output by using a literal string value that contains double quotes. C# is perfectly capable of working with such strings, as long as the string is prefixed with @ and use two sets of double quotes ("") in place of one set of double quotes. You must remember not to break the literal string into separate lines, unless the string you are comparing to is similarly broken. For example, the literal I use in the test method has wrapped onto two lines because the width of a printed page is narrow. I have not added a newline character; if I did, the test would fail.

An extension method is available for use only when the namespace that contains it is in scope. In a code file, this is done with a `using` statement; but for a Razor view, you must add a configuration entry to the `Web.config` file, or add a `@using` statement to the view itself. There are, confusingly, two `Web.config` files in a Razor MVC project: the main one, which resides in the root directory of the application project, and the view-specific one, which is in the `Views` folder. The change I want to make is to the `Views/web.config` file as shown in Listing 7-19.

Listing 7-19. Adding the HTML Helper Method Namespace to the Views/web.config File

```
...
  <system.web.webPages.razor>
    <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc, Version=5.0.0.0,
Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
    <pages pageBaseType="System.Web.Mvc.WebViewPage">
      <namespaces>
        <add namespace="System.Web.Mvc" />
        <add namespace="System.Web.Mvc.Ajax" />
        <add namespace="System.Web.Mvc.Html" />
        <add namespace="System.Web.Routing" />
        <add namespace="SportsStore.WebUI" />
        <add namespace="SportsStore.WebUI.HtmlHelpers"/>
      </namespaces>
    </pages>
  </system.web.webPages.razor>
...
```

Every namespace that I refer to in a Razor view needs to be used explicitly, declared in the `web.config` file or applied with a `@using` expression.

Adding the View Model Data

I am not quite ready to use the HTML helper method. I have yet to provide an instance of the `PagingInfo` view model class to the view. I could do this using the view bag feature, but I would rather wrap all of the data I am going to send from the controller to the view in a single view model class. To do this, I added a class file called

ProductsListViewModel.cs to the Models folder of the SportsStore.WebUI project. Listing 7-20 shows the contents of the new file.

Listing 7-20. The Contents of the ProductsListViewModel.cs File

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Models {
    public class ProductsListViewModel {

        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
    }
}
```

I can update the List action method in the ProductController class to use the ProductsListViewModel class to provide the view with details of the products to display on the page and details of the pagination, as shown in Listing 7-21.

Listing 7-21. Updating the List Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository productRepository) {
            this.repository = productRepository;
        }

        public ViewResult List(int page = 1) {
            ProductsListViewModel model = new ProductsListViewModel {
                Products = repository.Products
                .OrderBy(p => p.ProductID)
                .Skip((page - 1) * PageSize)
                .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = page,
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Count()
                }
            };
            return View(model);
        }
    }
}
```

These changes pass a `ProductsListViewModel` object as the model data to the view.

UNIT TEST: PAGE MODEL VIEW DATA

I need to ensure that the controller sends the correct pagination data to the view. Here is the unit test I added to the test project to test this:

```
...
[TestMethod]
public void Can_Send_Pagination_View_Model() {

    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    });

    // Arrange
    ProductController controller = new ProductController(mock.Object);
    controller.PageSize = 3;

    // Act
    ProductsListViewModel result = (ProductsListViewModel)controller.List(2).Model;

    // Assert
    PagingInfo pageInfo = result.PagingInfo;
    Assert.AreEqual(pageInfo.CurrentPage, 2);
    Assert.AreEqual(pageInfo.ItemsPerPage, 3);
    Assert.AreEqual(pageInfo.TotalItems, 5);
    Assert.AreEqual(pageInfo.TotalPages, 2);
}
...
```

I also need to modify the earlier pagination unit test, contained in the `Can_Paginate` method. It relies on the `List` action method returning a `ViewResult` whose `Model` property is a sequence of `Product` objects, but I have wrapped that data inside another view model type. Here is the revised test:

```
...
[TestMethod]
public void Can_Paginate() {

    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    });

    ProductController controller = new ProductController(mock.Object);
    controller.PageSize = 3;

    // Act
    ProductsListViewModel result = (ProductsListViewModel)controller.List(2).Model;

    // Assert
    Product[] prodArray = result.Products.ToArray();
    Assert.IsTrue(prodArray.Length == 2);
    Assert.AreEqual(prodArray[0].Name, "P4");
    Assert.AreEqual(prodArray[1].Name, "P5");
}
...
```

I would usually create a common setup method, given the degree of duplication between these two test methods. However, since I am delivering the unit tests in individual sidebars like this one, I am going to keep everything separate so you can see each test on its own.

The view is currently expecting a sequence of `Product` objects, so I need to update the `List.cshtml` file, as shown in Listing 7-22, to deal with the new view model type.

Listing 7-22. Updating the List.cshtml File

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
}
```

```
@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
```

I have changed the `@model` directive to tell Razor that I am now working with a different data type. I updated the `foreach` loop so that the data source is the `Products` property of the model data.

Displaying the Page Links

I have everything in place to add the page links to the `List` view. I created the view model that contains the paging information, updated the controller so that it passes this information to the view, and changed the `@model` directive to match the new model view type. All that remains is to call the HTML helper method from the view, which you can see in Listing 7-23.

Listing 7-23. Calling the HTML Helper Method in the List.cshtml File

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

<div>
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("List", new { page = x }))
</div>
```

If you run the application, you will see the new page links, as illustrated in Figure 7-14. The style is still basic, which I will fix later in the chapter. What is important for the moment is that the links take the user from page to page in the catalog and let him explore the products for sale.

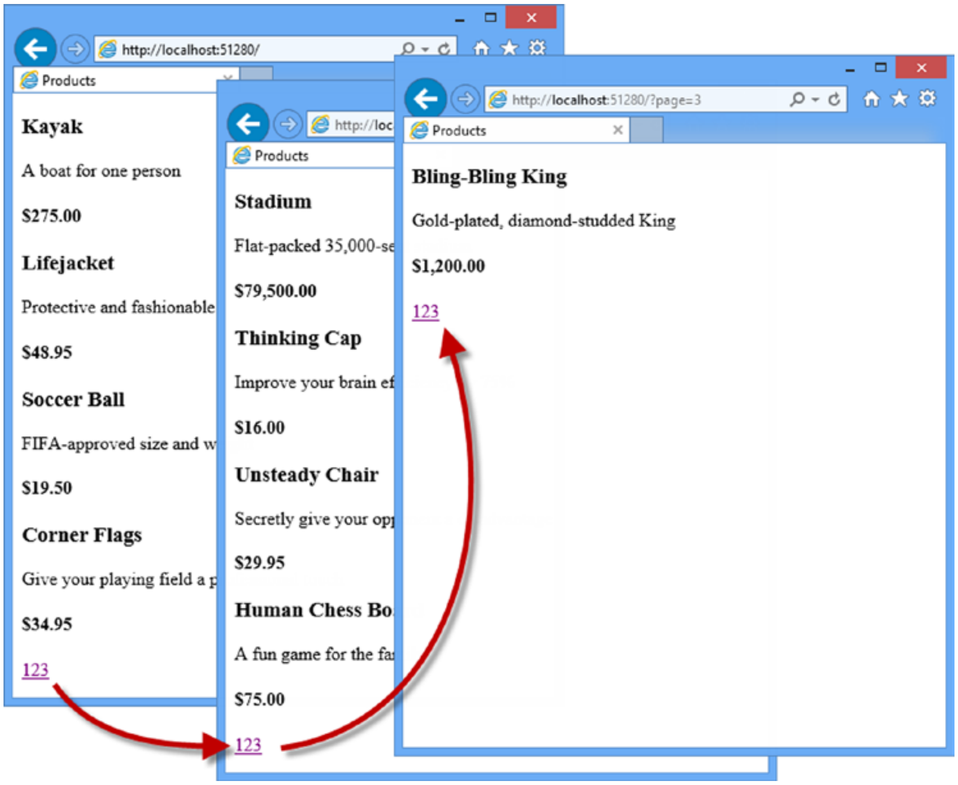


Figure 7-14. *Displaying page navigation links*

WHY NOT JUST USE A GRIDVIEW?

If you have worked with ASP.NET before, you might think that was a lot of work for an unimpressive result. It has taken me pages and pages just to get a page list. If I were using Web Forms, I could have done the same thing using the ASP.NET Web Forms `GridView` or `ListView` controls, right out of the box, by hooking it up directly to the `Products` database table.

What I have accomplished in this chapter may not look like much, but it is different from dragging a control onto a design surface. First, I am building an application with a sound and maintainable architecture that involves proper separation of concerns. Unlike the simplest use of the `ListView` control, I have not directly coupled the UI and the database: an approach that gives quick results but that causes pain and misery over time. Second, I have been creating unit tests as I go, and these allow me to validate the behavior of the application in a natural way that is nearly impossible with a complex Web Forms control. Finally, bear in mind that I have given over a lot of this chapter to creating the underlying infrastructure on which I am building the application. I need to define and implement the repository only once, for example, and now that I have, I will be able to build and test new features quickly and easily, as the following chapters will demonstrate.

None of this detracts from the immediate results that Web Forms can deliver, of course, but as I explained in Chapter 3, that immediacy comes with a cost that can be expensive and painful in large and complex projects.

Improving the URLs

I have the page links working, but they still use the query string to pass page information to the server, like this:

```
http://localhost/?page=2
```

I create URLs that are more appealing by creating a scheme that follows the pattern of *composable URLs*. A composable URL is one that makes sense to the user, like this one:

```
http://localhost/Page2
```

MVC makes it easy to change the URL scheme in application because it uses the ASP.NET *routing* feature. All I need do is add a new route to the `RegisterRoutes` method in the `RouteConfig.cs` file, which you will find in the `App_Start` folder of the `SportsStore.WebUI` project. You can see the change I made to this file in Listing 7-24.

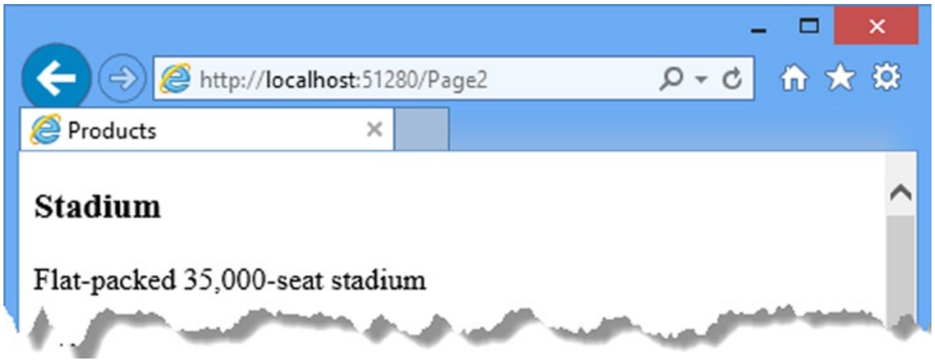
Listing 7-24. Adding a New Route to the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace SportsStore.WebUI {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: null,
                url: "Page{page}",
                defaults: new { Controller = "Product", action = "List" }
            );

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Product", action = "List",
                    id = UrlParameter.Optional }
            );
        }
    }
}
```

It is important that you add this route before the `Default` one that is already in the file. As you will see in Chapter 15, the routing system processes routes in the order they are listed, and I need the new route to take precedence over the existing one.

This is the only alteration required to change the URL scheme for product pagination. The MVC Framework and the routing function are tightly integrated, and so the application automatically reflects a change like this in the result produced by the `Url.Action` method (which is what I used in the `List.cshtml` view to generate the page links). Do not worry if routing does not make sense to you now. I explain it in detail in Chapters 15 and 16.

If you run the application and navigate to a page, you will see the new URL scheme in action, as illustrated in Figure 7-15.



Stadium

Flat-packed 35,000-seat stadium

Figure 7-15. *The new URL scheme displayed in the browser*

Styling the Content

I have built a great deal of infrastructure and the application is really starting to come together, but I have not paid any attention to its appearance. Even though this book is not about design or CSS, the SportsStore application design is so miserably plain that it undermines its technical strengths. In this section, I will put some of that right. I am going to implement a classic two-column layout with a header, as shown in Figure 7-16.

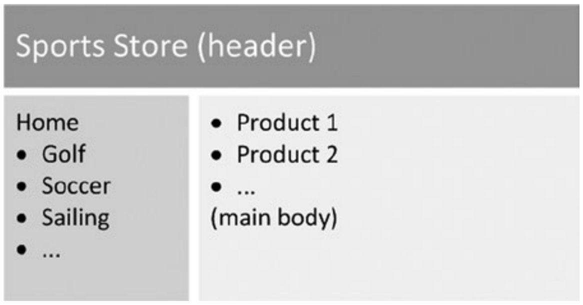


Figure 7-16. *The design goal for the SportsStore application*

Installing the Bootstrap Package

I am going to use the Bootstrap package to provide the CSS styles I will apply to the application. To install the Bootstrap package, select `Library Package Manager` ➤`Package Manager Console` from the Visual Studio `Tools` menu. Visual Studio will open the NuGet command line. Enter the following command and hit return:

```
Install-Package -version 3.0.0 bootstrap -projectname SportsStore.WebUI
```

This is the same basic NuGet command I used in Chapter 2, with the addition of the `projectname` argument to ensure NuGet adds the files to the right project.

■ **Note** Once again, I am going to use Bootstrap without going into the details of the features that the package provides. For full details of Bootstrap and the other client-side libraries that Microsoft has blessed for use with the MVC Framework, see my Pro ASP.NET MVC 5 Client book, published by Apress in 2014.

Applying Bootstrap Styles to the Layout

In Chapter 5, I explained how Razor layouts work and how you apply them. When I created the `List.cshtml` view for the `Product` controller, I asked you to check the option to use a layout, but leave the box that specifies a layout blank. This has the effect of using the layout specified by the `Views/_ViewStart.cshtml` file, which Visual Studio created automatically along the view. You can see the contents of the view start file in Listing 7-25.

Listing 7-25. The Contents of the _ViewStart.cshtml File

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

The value of the `Layout` property specifies that views will use the `Views/Shared/_Layout.cshtml` file as a layout, unless they explicitly specify an alternative. I reset the content of the `_Layout.cshtml` file earlier in the chapter to remove the template content that Visual Studio adds and in Listing 7-26 you can see how I have returned to this file to add the Bootstrap CSS file and apply some of the CSS styles it defines.

Listing 7-26. Applying Bootstrap CSS to the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div class="navbar navbar-inverse" role="navigation">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>
    <div class="row panel">
        <div id="categories" class="col-xs-3">
            Put something useful here later
        </div>
        <div class="col-xs-8">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

I have added the `bootstrap.css` and `bootstrap-theme.css` files to the layout using `link` elements and applied various Bootstrap classes to create a simple layout. I also need to change the `List.cshtml` file, as shown in Listing 7-27.

Listing 7-27. Applying Bootstrap to Style the List.cshtml File

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products) {
    <div class="well">
        <h3>
            <strong>@p.Name</strong>
            <span class="pull-right label label-primary">@p.Price.ToString("c")</span>
        </h3>
        <span class="lead"> @p.Description</span>
    </div>
}

<div class="btn-group pull-right">
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("List", new { page = x }))
</div>
```

THE PROBLEM WITH STYLING ELEMENTS

The HTML elements generated by an MVC application come from a variety of sources (static content, Razor expressions, HTML helper methods, etc.), so the style classes become diffused throughout the project. If this makes you feel slightly uncomfortable, then you are not alone. Mixing the CSS styles in with element generation is not a great idea and runs counter to the idea of separating out unrelated functionality that pervades MVC. You can improve on this situation by assigning non-Bootstrap classes to elements based on their role in the application and then use a library like jQuery or LESS to map between your custom classes and the Bootstrap ones.

I am going to keep things simple for this application and accept that I have embedded the Bootstrap classes throughout the application, even though it complicates the process of changing styles in the future. I would not do this in a real project, but I know this example application is not going to enter into a maintenance phase.

If you run the application, you will see that I have improved the appearance—at least a little, anyway—as illustrated by Figure 7-17.

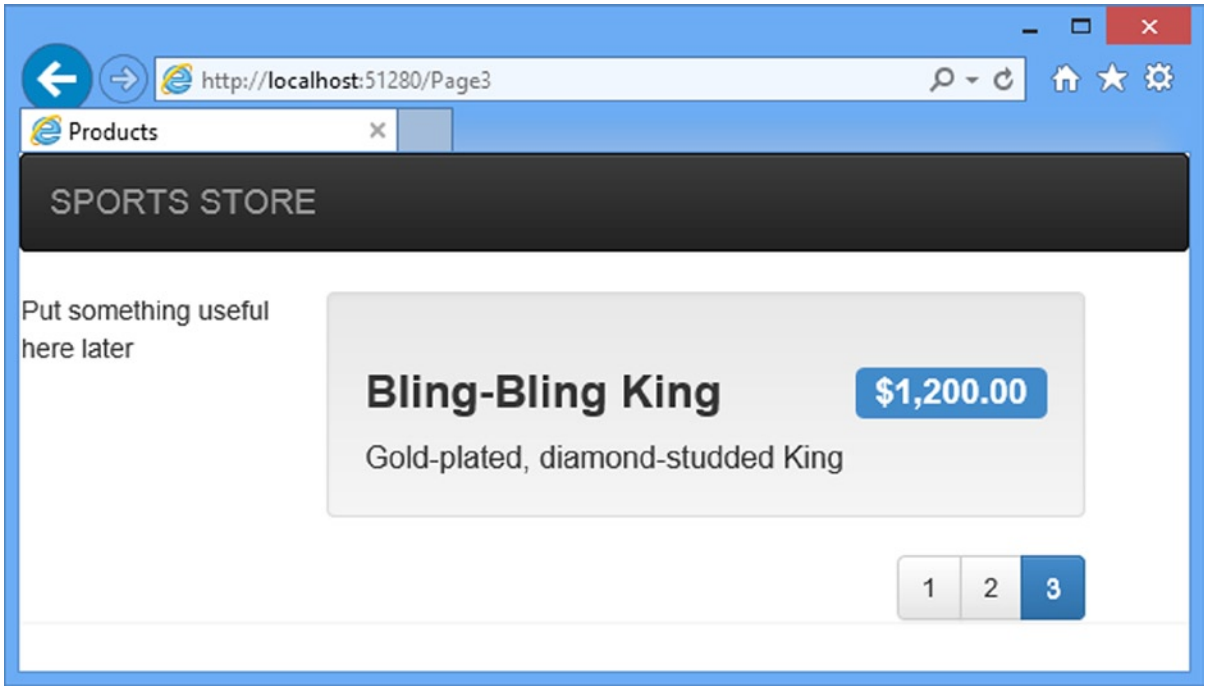
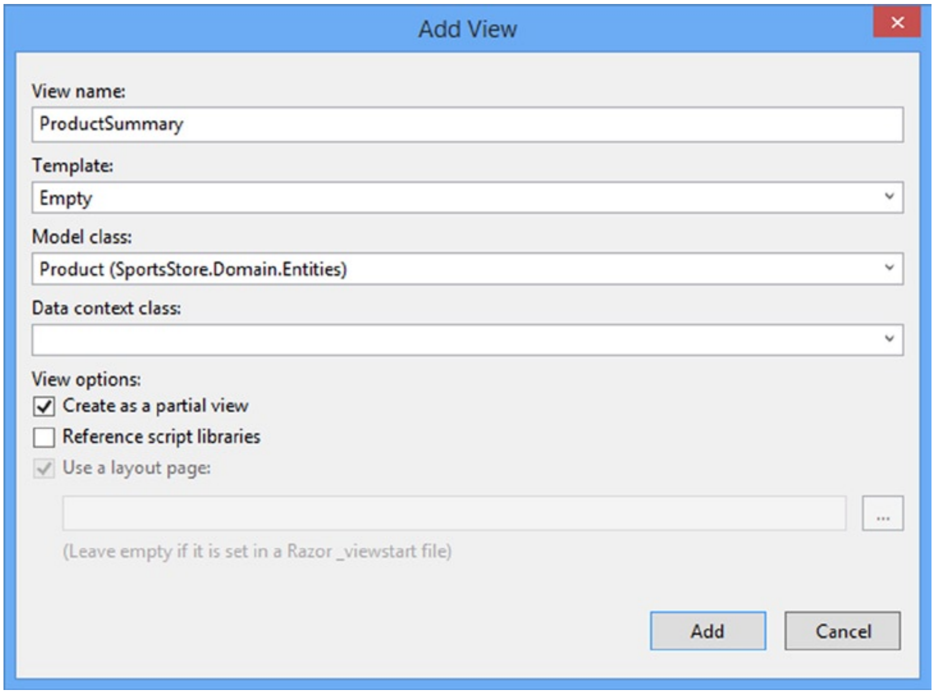

http://localhost:51280/Page3
Products
SPORTS STORE
Put something useful here later
Bling-Bling King
$1,200.00
Gold-plated, diamond-studded King
1
2
3

Figure 7-17. *The design-enhanced SportsStore application*

Creating a Partial View

As a finishing trick for this chapter, I am going to refactor the application to simplify the `List.cshtml` view. I am going to create a *partial view*, which is a fragment of content that you can embed into another view, rather like a template. Partial views are contained within their own files and are reusable across multiple views, which can help reduce duplication if you need to render the same kind of data in several places in your application.

To add the partial view, right-click the `/Views/Shared` folder in the `SportsStore.WebUI` project and select `Add` ➤ `View` from the pop-up menu. Set `View Name` to `ProductSummary`, set `Template` to `Empty`, select `Product` from the `Model Class` drop-down list and check the `Create As A Partial View` box, as shown in Figure 7-18.



View name:
ProductSummary
Template:
Empty
Model class:
Product (SportsStore.Domain.Entities)
Data context class:

View options:
[✓] Create as a partial view
[ ] Reference script libraries
[✓] Use a layout page:
(Leave empty if it is set in a Razor _viewstart file)

Figure 7-18. *Creating a partial view*

Click the Add button, and Visual Studio will create a partial view file called `Views/Shared/ProductSummary.cshtml`. A partial view is similar to a regular view, except that it produces a fragment of HTML, rather than a full HTML document. If you open the `ProductSummary` view, you will see that it contains only the model view directive, which is set to the `Product` domain model class. Apply the changes shown in Listing 7-28.

Listing 7-28. Adding Markup to the ProductSummary.cshtml File

```
@model SportsStore.Domain.Entities.Product

<div class="well">
    <h3>
        <strong>@Model.Name</strong>
        <span class="pull-right label label-primary">@Model.Price.ToString("c")</span>
    </h3>
    <span class="lead"> @Model.Description</span>
</div>
```

Now I need to update `Views/Products/List.cshtml` so that it uses the partial view. You can see the change in Listing 7-29.

Listing 7-29. Using a Partial View in the List.cshtml File

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products) {
    @Html.Partial("ProductSummary", p)
}

<div class="pager">
   @Html.PageLinks(Model.PagingInfo, x => Url.Action("List", new {page = x}))
</div>
```

I have taken the markup that was previously in the `foreach` loop in the `List.cshtml` view and moved it to the new partial view. I call the partial view using the `Html.Partial` helper method. The parameters are the name of the view and the view model object. Switching to a partial view like this is good practice, but it does not change the appearance of the application, as Figure 7-19 shows.

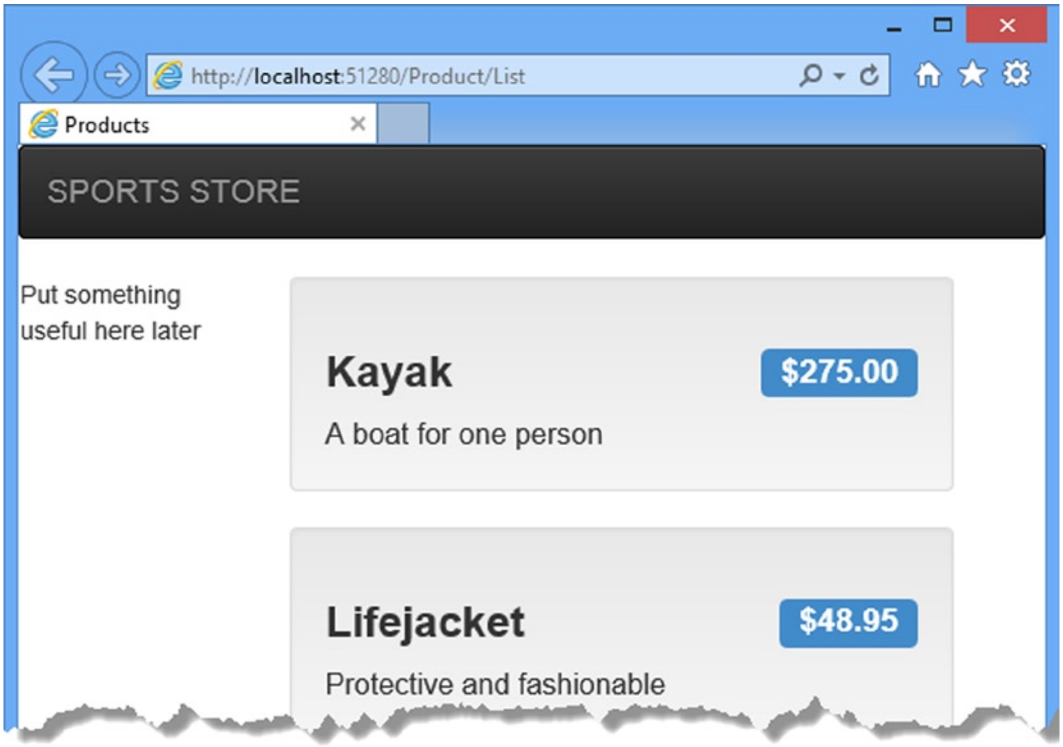

http://localhost:51280/Product/List
Products
SPORTS STORE
Put something useful here later
Kayak
$275.00
A boat for one person
Lifejacket
$48.95
Protective and fashionable

Figure 7-19. *Applying a partial view*

Summary

In this chapter, I built most of the core infrastructure for the SportsStore application. It does not have many features that you could demonstrate to a client at this point, but behind the scenes, there are the beginnings of a domain model with a product repository backed by SQL Server and the Entity Framework. There is a single controller, `ProductController`, that can produce paginated lists of products, and I have set up DI and defined a clean and friendly URL scheme.

If this chapter felt like a lot of setup for little benefit, then the next chapter will balance the equation. Now that the fundamental structure is in place, we can forge ahead and add all of the customer-facing features: navigation by category, a shopping cart, and a checkout process.