

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 development is like by creating as realistic an example as possible. I want to focus on the ASP.NET Core MVC, of course, 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, 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 how unit testing can be an important and useful 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 Core MVC, although, of course, support for testing is a key reason for adopting ASP.NET Core MVC.

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. See the instructions earlier in this update.

Note If you just want to follow the project without having to re-create it, then you can download the SportsStore project as part of the free source code download that accompanies this book available at 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 MVC Project

I am going to follow the same basic approach that I used in earlier chapters, which is to start with an empty project and add all of the configuration files and components that I require. I started by selecting New > Project from the Visual Studio File menu and selecting the ASP.NET Core Web Application (.NET Core) project template, as shown in Figure 8-1. I set the name of the project to be SportsStore and clicked the OK button.

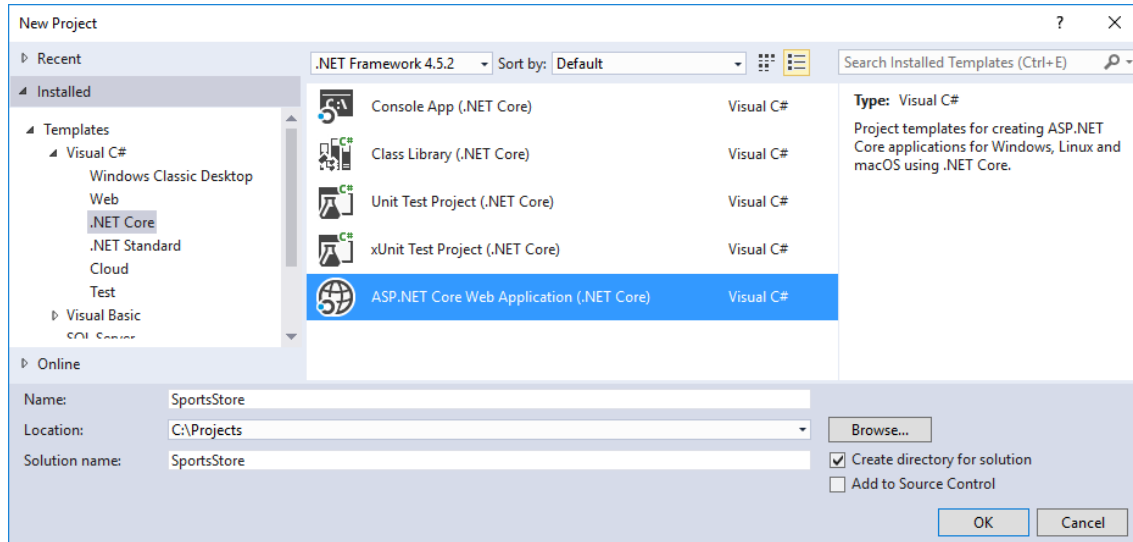


Figure 8-1. Selecting the project type

I selected ASP.NET Core 1.1 in the drop-down list and selected the Empty template, as shown in Figure 8-2. I clicked the OK button to create the SportsStore project.

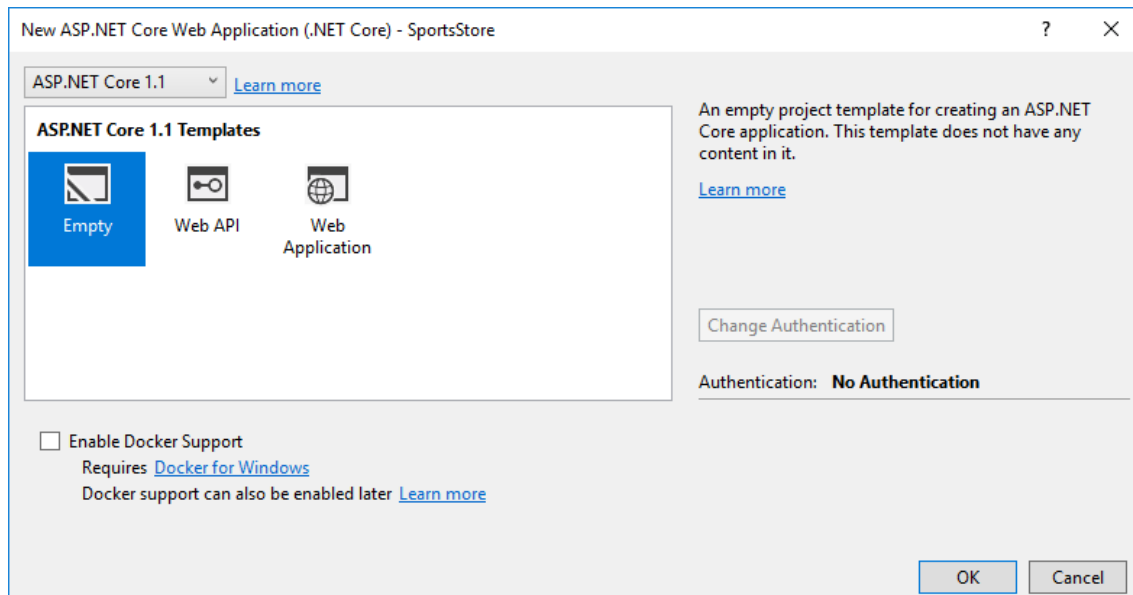


Figure 8-2. Selecting the project template

Adding the NuGet Packages

With the introduction of .NET Core 1.1 and Visual Studio 2017, Microsoft has changed the way that NuGet packages are added to projects. The `project.json` file from the Visual Studio 2015 era has been replaced by the `SportsStore.csproj` file, which does the same basic job but uses an XML format.

To edit the `csproj` file, right click on the SportsStore project item in the Solution Explorer and select `Edit SportsStore.csproj` from the popup menu.

The Empty project template installs the basic ASP.NET Core features but requires additional packages to provide functionality required for MVC applications. Listing 8-1 shows the additions I made to the `SportsStore.csproj` file to add the packages that I need to get started with the SportsStore application.

Tip Microsoft has released minor updates to the ASP.NET Core and ASP.NET Core MVC packages since Pro ASP.NET Core MVC was published. I have used these revised versions in this update but the examples work in exactly the same way.

Listing 8-1. Adding NuGet Packages in the SportsStore.csproj File

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore"
      Version="2.0.0" />
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
    <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.1" />
    <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink"
      Version="1.1.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
      Version="1.0.0" />
  </ItemGroup>
</Project>
```

```
</ItemGroup>

</Project>
```

When you save the changes to the file, Visual Studio will download and install the new packages. Packages that are required to run the project are added using **PackageReference** elements, with the **Include** attribute used to specify the package name and the **Version** attribute used to specify the version that is required. Packages that are used to set up tooling, equivalent to the **tools** section of the **project.json** file, are added using **DotNetCliToolReference** elements.

Note Microsoft provides command line tools for adding packages to projects, as well as including support for managing packages visually within Visual Studio. However, these tools do not yet have the ability to manage tooling packages - the ones that require **DotNetCliToolReference** elements - and so editing the file is the simplest way to configure a project.

The packages added to the project provide the basic functionality required to get started with MVC development. I'll add other packages as the SportsStore application develops, but these packages are a good starting point, as described in Table 8-1.

Table 8-1. The Essential NuGet Packages for MVC Development

Name	Description
Microsoft.AspNetCore.Mvc	This package contains ASP.NET Core MVC and provides access to essential features such as controllers and Razor views.
Microsoft.AspNetCore.StaticFiles	This package provides support for serving static files, such as images, JavaScript, and CSS, from the wwwroot folder.
Microsoft.VisualStudio.Web.BrowserLink	This package provides support for automatically reloading the browser when files in the project change, which can be a useful feature during development.

Creating the Folder Structure

The next step is to add the folders that will contain the application components required for an MVC application: models, controllers, and views. For each of the folders described in Table 8-2, right-click the SportsStore project item in the Solution Explorer (the item inside the `src` folder), select Add > New Folder from the pop-up menu, and set the folder name. Additional folders will be required later, but these reflect the main parts of the MVC application and are enough to get started with.

Table 8-2. *The Folders Required for the SportsStore Project*

Name	Description
Models	This folder will contain the model classes.
Controllers	This folder will contain the controller classes.
Views	This folder holds everything related to views, including individual Razor files, the view start file, and the view imports file.

Configuring the Application

An ASP.NET Core MVC application relies on several configuration files. First, having installed the NuGet packages, I need to edit the `Startup` class to tell ASP.NET to use them, as shown in Listing 8-2.

Listing 8-2. Enabling Features in the Startup.cs File

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

namespace SportsStore {

    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
            IHostingEnvironment env, ILoggerFactory loggerFactory) {
            app.UseDeveloperExceptionPage();
        }
    }
}
```

```
        app.UseStatusCodePages();
        app.UseStaticFiles();
        app.UseMvcWithDefaultRoute();
    }
}
```

The **ConfigureServices** method is used to set up shared objects that can be used throughout the application through the dependency injection feature, which I describe in Chapter 18. The **AddMvc** method that I call in the **ConfigureServices** method is an extension method that sets up the shared objects used in MVC applications.

The **Configure** method is used to set up the features that receive and process HTTP requests. Each method that I call in the **Configure** method is an extension method that sets up an HTTP request processor, as described in Table 8-3.

Note The **Startup** class is an important ASP.NET Core feature. I describe it in detail in Chapter 14.

Table 8-3. *The Initial Feature Methods Called in the Start Class*

Method	Description
UseDeveloperExceptionPage()	This extension method displays details of exceptions that occur in the application, which is useful during the development process. It should not be enabled in deployed applications, and I disable this feature in Chapter 12.
UseStatusCodePages()	This extension method adds a simple message to HTTP responses that would not otherwise have a body, such as 404 - Not Found responses.
UseStaticFiles()	This extension method enables support for serving static content from the wwwroot folder.
UseMvcWithDefaultRoute()	This extension method enables ASP.NET Core MVC with a default configuration (which I will change later in the development process).

Next, I need to prepare the application for Razor views. Right-click the Views folder, select Add > New Item from the pop-up menu, and select the MVC View Imports Page item from the ASP.NET Core > Web > ASP.NET category, as shown in Figure 8-3.

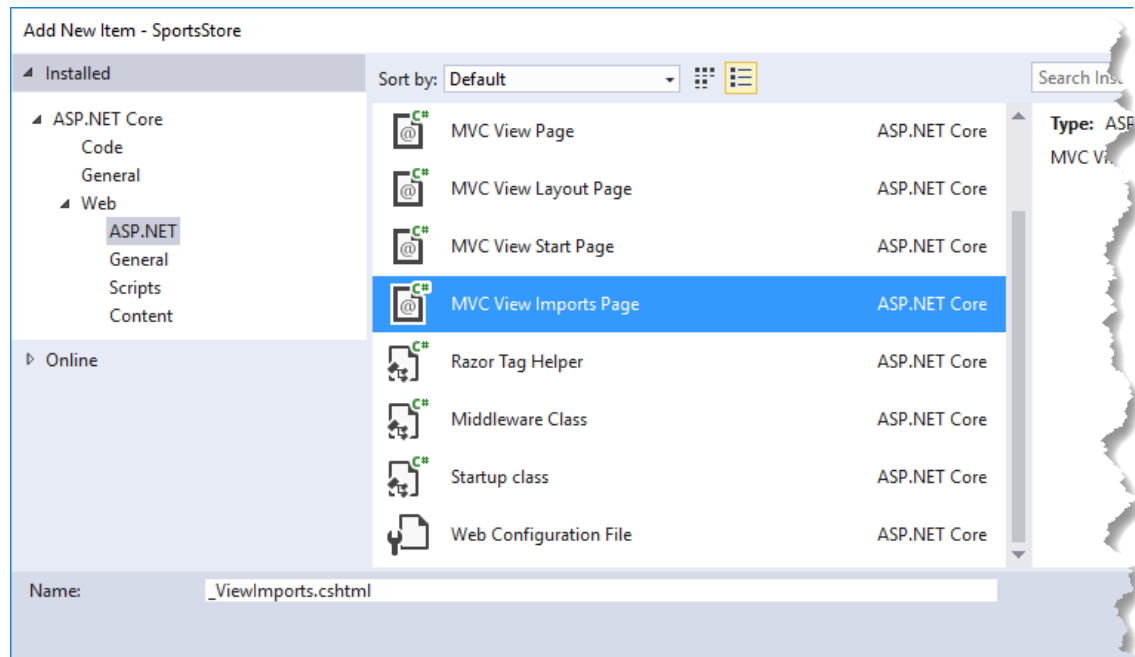


Figure 8-3. Creating the view imports file

Click the Add button to create the `_ViewImports.cshtml` file and set the contents of the new file to match Listing 8-3.

Listing 8-3. The Contents of the `_ViewImports.cshtml` File in the Views Folder

```
@using SportsStore.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The `@using` statement will allow me to use the types in the `SportsStore.Models` namespace in views without needing to refer to the namespace. The `@addTagHelper` statement enables the built-in tag helpers, which I use later to create HTML elements that reflect the configuration of the SportsStore application.

Creating the Unit Test Project

The process for creating unit test projects has been simplified in Visual Studio 2017. Right-click on the SportsStore solution item in the Solution Explorer and select Add > New Project from the popup menu. Select **xUnit Test Project (.NET Core)** from the list of project templates, as

shown in Figure 8-4 and set the name of the project to **SportsStore.Tests**. Click OK to create the unit test project.

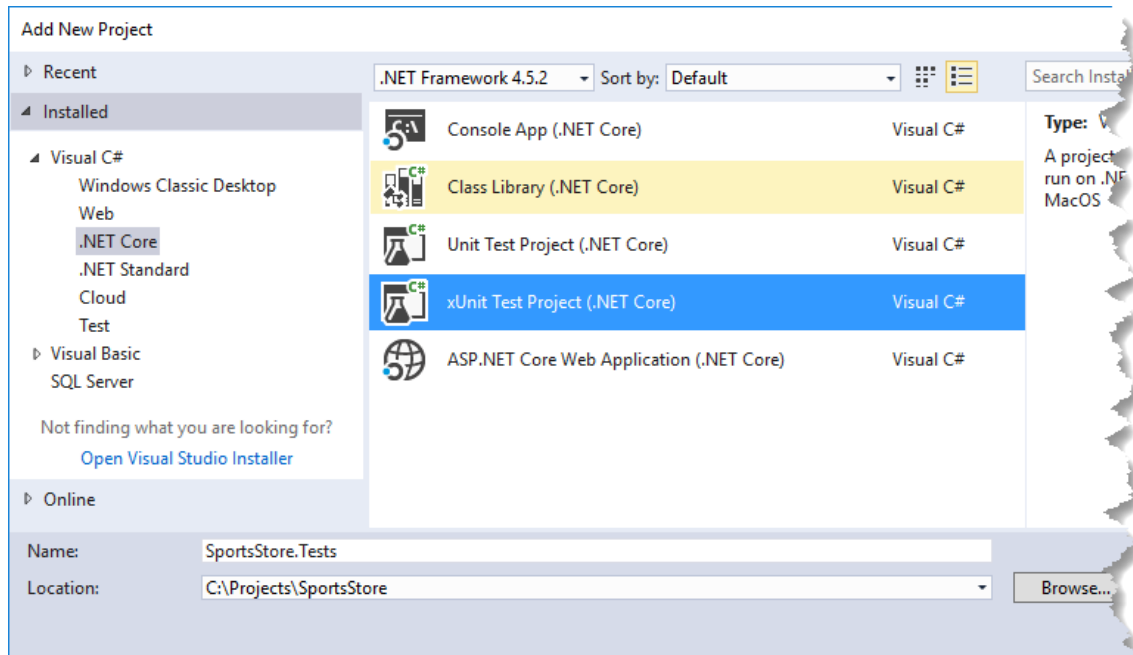


Figure 8-4. Creating the unit test project

Once the unit test project has been created, right-click the **SportsStore.Tests** project item in the Solution Explorer and select **Edit SportsStore.Tests.csproj** from the popup window. Add the elements shown in Listing 8-4 to configure the project.

Listing 8-4. The Contents of the SportsStore.Tests.csproj File in the Unit Test Project

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\SportsStore\SportsStore.csproj" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.0.0" />
  </ItemGroup>
</Project>
```

```
<PackageReference Include="xunit" Version="2.2.0" />
<PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
<PackageReference Include="Moq" Version="4.7.1" />
</ItemGroup>

</Project>
```

The **ProjectReference** element adds a dependency on the main SportsStore project, while the **PackageReference** elements list the NuGet packages that are required for unit testing. Save the changes and Visual Studio will download and install the NuGet packages.

Note In the original book chapters, I used a version of the Moq package that had been prepared by Microsoft and which required a configuration change in Visual Studio. This is no longer needed, since the main Moq package now works with .NET Core.

Checking and Running the Application

The application and unit test projects are created and configured and ready for development. The Solution Explorer should contain the items shown in Figure 8-5. You will have problems if you see different items or items are not in the same locations, so take a moment to check that everything is present and in the right place.

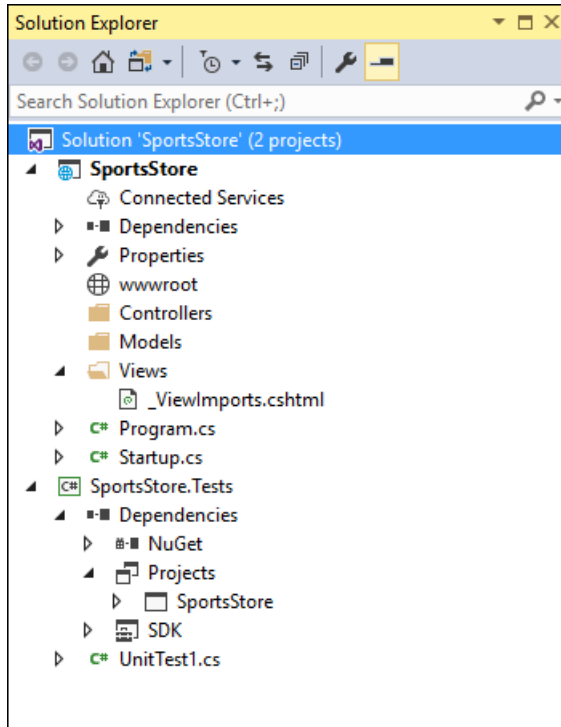


Figure 8-5. The Solution Explorer for the SportsStore application and unit test projects

If you select Start Debugging from the Debug menu (or Start Without Debugging if you prefer the iterative development style I described in Chapter 6), you will see an error page, as shown in Figure 8-6. The error message is shown because there are no controllers in the application to handle requests at the moment, which is something that I will address shortly.

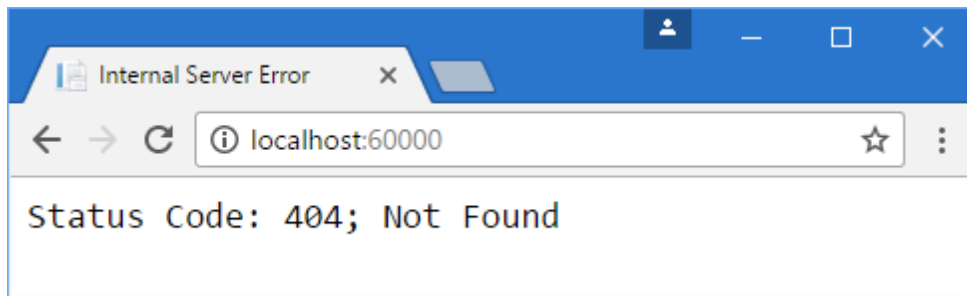


Figure 8-6. Running the SportsStore application

Starting the Domain Model

All projects start with the domain model, which is the heart of an MVC application. Since this is an e-commerce application, the most obvious model I need is for a product. I added a class file called **Product.cs** to the **Models** folder and used it to define the class shown in Listing 8-5.

Listing 8-5. The Contents of the Product.cs File in the Models Folder

```
namespace SportsStore.Models {  
  
    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; }  
    }  
}
```

Creating a Repository

I need some way of getting **Product** objects from a database. As I explained in Chapter 3, the model includes the logic for storing and retrieving the data from the persistent data store. I won't 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. I added a new C# interface file called **IProductRepository.cs** to the **Models** folder and used it to define the interface shown in Listing 8-6.

Listing 8-6. The Contents of the IProductRepository.cs File in the Models Folder

```
using System.Collections.Generic;  
  
namespace SportsStore.Models {  
  
    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. I will revisit the `IProductRepository` interface throughout the development process to add features.

Creating a Fake Repository

Now that I have defined an 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. To do this, I am going to create a fake implementation of the `IProductRepository` interface that will stand in until I return to the topic of data storage. To create the fake repository, I added a class file called `FakeProductRepository.cs` to the `Models` folder and used it to define the class shown in Listing 8-7.

Listing 8-7. The Contents of FakeProductRepository.cs File in the Models Folder

```
using System.Collections.Generic;

namespace SportsStore.Models {

    public class FakeProductRepository : IProductRepository {

        public IEnumerable<Product> Products => new List<Product> {
            new Product { Name = "Football", Price = 25 },
            new Product { Name = "Surf board", Price = 179 },
            new Product { Name = "Running shoes", Price = 95 }
        };
    }
}
```

The `FakeProductRepository` class implements the `IProductRepository` interface by returning a fixed collection of `Product` objects as the value of the `Products` property.

Registering the Repository Service

MVC emphasizes the use of *loosely coupled components*, which means that you can make a change in one part of the application without having to make corresponding changes elsewhere. This approach categorizes parts of the application as *services*, which provide features that other parts of the application use. The class that provides a service can then be altered or replaced without requiring changes in the classes that use it. I explain this in depth in Chapter 18 but for the SportsStore application, I want to create a repository service, which allows controllers to get objects that implement the `IProductRepository` interface without knowing which class is being used. This will allow me to start developing the application using

the simple **FakeProductRepository** class I created in the previous section and then replace it with a real repository later without having to make changes in all of the classes that need access to the repository. Services are registered in the **ConfigureServices** method of the **Startup** class, and in Listing 8-8, I have defined a new service for the repository.

Listing 8-8. Creating the Repository Service in the Startup.cs File

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using SportsStore.Models;

namespace SportsStore {

    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddTransient<IProductRepository,
                FakeProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
            IHostingEnvironment env, ILoggerFactory loggerFactory) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

The statement I added to the **ConfigureServices** method tells ASP.NET that when a component, such as a controller, needs an implementation of the **IProductRepository** interface, it should receive an instance of the **FakeProductRepository** class. The **AddTransient** method specifies that a new **FakeProductRepository** object should be created each time the **IProductRepository** interface is needed. Don't worry if this doesn't make sense at the moment; you will see how it fits into the application shortly, and I explain what is happening in detail in Chapter 18.

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 rest of the application at all. I think you would find that boring, though, so I am going to switch tracks and start using MVC in earnest and come back to 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 fake 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

To create the first controller in the application, I added a class file called **ProductController.cs** to the **Controllers** folder and defined the class shown in Listing 8-9.

Listing 8-9. The Contents of the ProductController.cs File

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {


    public class ProductController : Controller {
        private IProductRepository repository;

        public ProductController(IProductRepository repo) {
            repository = repo;
        }
    }
}
```

When MVC needs to create a new instance of the **ProductController** class to handle an HTTP request, it will inspect the constructor and see that it requires an object that implements the **IProductRepository** interface. To determine what implementation class should be used, MVC consults the configuration in the **Startup** class, which tells it that **FakeRepository** should be used and that a new instance should be created every time. MVC creates a new **FakeRepository** object and uses it to invoke the **ProductController** constructor in order to create the controller object that will process the HTTP request.

This is known as *dependency injection*, and its approach allows the **ProductController** to access the application's repository through the **IProductRepository** interface without having

any need to know which implementation class has been configured. Later, I'll replace the fake repository with the real one, and dependency injection means that the controller will continue to work without changes.

 **Note** Some developers don't like dependency injection and believe it makes applications more complicated. That's not my view, but if you are new to dependency injection, then I recommend you wait until you have read Chapter 18 before you make up your mind.

Next, I have added an action method, called **List**, which will render a view showing the complete list of the products in the repository, as shown in Listing 8-10.

Listing 8-10. Adding an Action Method in the ProductController.cs File

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;

        public ProductController(IProductRepository repo) {
            repository = repo;
        }

        public IActionResult List() => View(repository.Products);
    }
}
```

Calling the **View** method like this (without specifying a view name) tells MVC to render the default view for the action method. Passing a **List<Product>** (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 and Configuring the View

I need to create a view to present the content to the user, but there are some preparatory steps required that will make writing the view simpler. The first is to create a shared layout that will define common content that will be included in all HTML responses sent to clients.

Shared layouts are a useful way of ensuring that views are consistent and contain important JavaScript files and CSS stylesheets, and I explained how they worked in Chapter 5.

I created the **Views/Shared** folder and added to it a new MVC view layout page called **_Layout.cshtml**, which is the default name that Visual Studio assigns to this item type. Listing 8-11 shows the **_Layout.cshtml** file. I made one change to the default content, which is to set the contents of the **title** element to **SportsStore**.

Listing 8-11. The Contents of the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Next, I need to configure the application so that the **_Layout.cshtml** file is applied by default. This is done by adding an MVC View Start Page file called **_ViewStart.cshtml** to the **Views** folder. The default content added by Visual Studio, shown in Listing 8-12, selects a layout called **_Layout.cshtml**, which corresponds to the file shown in Listing 8-11.

Listing 8-12. The Contents of the _ViewStart.cshtml File in the Views Folder

```
@{
    Layout = "_Layout";
}
```

Now I need to add the view that will be displayed when the **List** action method is used to handle a request. I created the **Views/Product** folder and added to it a Razor view file called **List.cshtml**. I then added the markup shown in Listing 8-13.

Listing 8-13. The Contents of the List.cshtml File in the Views/Product Folder

```
@model IEnumerable<Product>

@foreach (var p in Model) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
    </div>
}
```

```

        <h4>@p.Price.ToString("c")</h4>
    </div>
}

```

The `@model` expression at the top of the file specifies that the view will receive a sequence of `Product` objects from the action method as its model data. I use a `@foreach` expression to work through the sequence and generate a simple set of HTML elements for each `Product` object that is received.

The view doesn't know where the `Product` objects came from, how they were obtained, or whether or not they represent all of the products known to the application. Instead, the view deals only with how details of each `Product` is displayed using HTML elements, which is consistent with the separation of concerns that I described in Chapter 3.

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`.

Setting the Default Route

I need to tell MVC 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 `Startup` class that sets up the MVC classes that handle HTTP requests, as shown in Listing 8-14.

Listing 8-14. Changing the Default Route in the Startup.cs File

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using SportsStore.Models;

namespace SportsStore {

    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddTransient<IProductRepository,

```

```

        FakeProductRepository>();
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app,
        IHostingEnvironment env, ILoggerFactory loggerFactory) {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
        app.UseStaticFiles();
        app.UseMvc(routes => {
            routes.MapRoute(
                name: "default",
                template: "{controller=Product}/{action=List}/{id?}");
        });
    }
}

```

The **Configure** method of the **Startup** class is used to set up the request pipeline, which consists of classes (known as *middleware*) that will inspect HTTP requests and generate responses. The **UseMvc** method sets up the MVC middleware, and one of the configuration options is the scheme that will be used to map URLs to controllers and action methods. I describe the routing system in detail in Chapters 15 and 16, but the change in Listing 8-14 tells MVC to send requests to the **List** action method of the **Product** controller unless the request URL specifies otherwise.

Tip Notice that I have set the name of the controller in Listing 8-14 to be **Product** and not **ProductController**, which is the name of the class. This is part of the MVC naming convention, in which controller class names generally end in **Controller**, but you omit this part of the name when referring to the class. I explain the naming convention and its effect in Chapter 31.

Running the Application

All the basics are in place. I have a controller with an action method that MVC will use when the default URL for the application is requested. MVC will create an instance of the **FakeRepository** class and use it to create a new controller object to handle the request. The fake repository will provide the controller with some simple test data, which its action method passed to the Razor view so that the HTML response to the browser includes details for each product. When generating the HTML response, MVC will combine the data from the view selected by the action method with the content from the shared layout, producing a complete

HTML document that the browser can parse and display. You can see the result by starting the application, as shown in Figure 8-7.

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

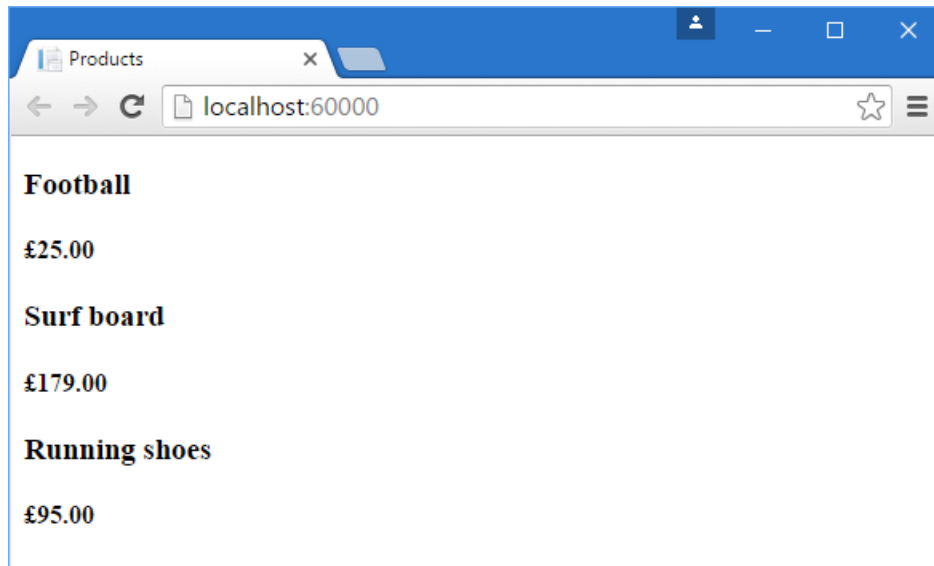


Figure 8-7. Viewing the basic application functionality

Preparing a Database

I can display a simple view that contains details of the products, but it uses the test data that the fake repository contains. Before I can implement a real repository with real data, 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 Core (EF Core), which is the Microsoft .NET object-relational mapping (ORM) framework. An ORM framework presents the tables, columns, and rows of a relational database through regular C# objects.

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; these variations may give you a better fit for your projects.

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

A nice feature of Visual Studio and SQL Server is *LocalDB*, which is an administration-free implementation of the basic 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 later. Most MVC applications are deployed to hosted environments that are run by professional administrators, so the LocalDB feature means that database configuration can be left in the hands of DBAs and developers can get on with coding.

Tip LocalDB is included as part of the workload selected when you installed Visual Studio. No additional download or configuration is required.

Installing Entity Framework Core

Entity Framework Core is installed using NuGet, and Listing 8-15 shows the additions that are required to the **SportsStore.csproj** file in the SportsStore application project.

Tip The **Folder** items that are shown in the listing were added by the MVC packages. The **SportsStore.csproj** file is rewritten by different parts of Visual Studio and individual packages, which means that you will often find elements that you did not explicitly add.

Listing 8-15. Adding Entity Framework in the SportsStore.csproj File in the SportsStore Project

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
```

```

    <TargetFramework>netcoreapp1.1</TargetFramework>
</PropertyGroup>

<ItemGroup>
  <Folder Include="Views\Shared\" />
  <Folder Include="wwwroot\" />
</ItemGroup>

<ItemGroup>
  <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore"
    Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.1" />
  <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink"
    Version="1.1.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="1.1.1" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
    Version="1.1.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
    Version="1.1.1" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="1.1.1" />

  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
    Version="1.0.0" />
</ItemGroup>

</Project>

```

Databases are managed using command-line tools, which are set up using a [DotNetCliToolReference](#) element in the [SportsStore.csproj](#) file, as shown in Listing 8-16.

Tip Visual Studio doesn't always reflect the changes in the [csproj](#) files, which means that you might see red underlying in your code files for packages you have recently added. Restart Visual Studio and open the solution again to fix the problem.

Listing 8-16. Registering the EF Core Tools in the SportsStore.csproj File in the SportsStore Project

```

<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

```

```

<ItemGroup>
  <Folder Include="Views\Shared\" />
  <Folder Include="wwwroot\" />
</ItemGroup>

<ItemGroup>
  <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore"
    Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.1" />
  <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink"
    Version="1.1.0" />

  <PackageReference Include="Microsoft.EntityFrameworkCore" Version="1.1.1" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
    Version="1.1.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
    Version="1.1.1" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="1.1.1" />

  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
    Version="1.0.0" />
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
    Version="1.0.0" />
</ItemGroup>
</Project>

```

When you save the **SportsStore.csproj** file, Visual Studio will download and install EF Core and add it to the project.

Creating the Database Classes

The *database context class* is the bridge between the application and the EF Core and provides access to the application's data using model objects. To create the database context class for the SportsStore application, I added a class file called **ApplicationDbContext.cs** to the **Models** folder and defined the class shown in Listing 8-17.

Listing 8-17. The Contents of the ApplicationDbContext.cs File in the Models Folder

```

using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {
    public class ApplicationDbContext : DbContext {

```

```

        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options) {}

        public DbSet<Product> Products { get; set; }
    }
}

```

The **DbContext** base class provides access to the Entity Framework Core's underlying functionality, and the **Products** property will provide access to the **Product** objects in the database. To populate the database and provide some sample data, I added a class file called **SeedData.cs** to the **Models** folder and defined the class shown in Listing 8-18.

Listing 8-18. The Contents of the SeedData.cs File in the Models Folder

```

using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace SportsStore.Models {

    public static class SeedData {

        public static void EnsurePopulated(IApplicationBuilder app) {
            ApplicationDbContext context = app.ApplicationServices
                .GetRequiredService<ApplicationDbContext>();
            if (!context.Products.Any()) {
                context.Products.AddRange(
                    new Product {
                        Name = "Kayak", Description = "A boat for one person",
                        Category = "Watersports", Price = 275 },
                    new Product {
                        Name = "Lifejacket",
                        Description = "Protective and fashionable",
                        Category = "Watersports", Price = 48.95m },
                    new Product {
                        Name = "Soccer Ball",
                        Description = "FIFA-approved size and weight",
                        Category = "Soccer", Price = 19.50m },
                    new Product {
                        Name = "Corner Flags",
                        Description = "Give your playing field a professional touch",
                        Category = "Soccer", Price = 34.95m },
                    new Product {
                        Name = "Stadium",
                        Description = "Flat-packed 35,000-seat stadium",
                        Category = "Soccer", Price = 79500 },
                    new Product {

```



```

        Name = "Thinking Cap",
        Description = "Improve brain efficiency by 75%",
        Category = "Chess", Price = 16 },
    new Product {
        Name = "Unsteady Chair",
        Description = "Secretly give your opponent a disadvantage",
        Category = "Chess", Price = 29.95m },
    new Product {
        Name = "Human Chess Board",
        Description = "A fun game for the family",
        Category = "Chess", Price = 75 },
    new Product {
        Name = "Bling-Bling King",
        Description = "Gold-plated, diamond-studded King",
        Category = "Chess", Price = 1200
    }
    };
    context.SaveChanges();
}
}
}
}
}

```

The static **EnsurePopulated** method receives an **IApplicationBuilder** argument, which is the class used in the **Configure** method of the **Startup** class to register middleware classes to handle HTTP requests, which is where I will ensure that the database has content.

The **EnsurePopulated** method obtains an **ApplicationDbContext** object through the **IApplicationBuilder** interface and uses it to check whether there are any **Product** objects in the database. If there are no objects, then the database is populated using a collection of **Product** objects using the **AddRange** method and then written to the database using the **SaveChanges** method.

Creating the Repository Class

It may not seem like it at the moment, but most of the work required to set up the database is complete. The next step is to create a class that implements the **IProductRepository** interface and gets its data using Entity Framework Core. I added a class file called **EFProductRepository.cs** to the **Models** folder and used it to define the repository class shown in Listing 8-19.

Listing 8-19. The Contents of the EFProductRepository.cs File in the Models Folder

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

```
namespace SportsStore.Models {

    public class EFProductRepository : IProductRepository {
        private ApplicationDbContext context;

        public EFProductRepository(ApplicationDbContext ctx) {
            context = ctx;
        }

        public IEnumerable<Product> Products => context.Products;
    }
}
```

I'll add additional functionality as I add features to the application, but for the moment, the repository implementation just maps the **Products** property defined by the **IProductRepository** interface onto the **Products** property defined by the **ApplicationDbContext** class.

Defining the Connection String

A *connection string* specifies the location and name of the database and provides configuration settings for how the application should connect to the database server. Connection strings are stored in a JSON file called **appsettings.json**, which I created in the SportsStore project using the ASP.NET Configuration File item template in the ASP.NET section of the Add New Item window.

Visual Studio adds a placeholder connection string to the **appsettings.json** file when it creates the file, which I have edited in Listing 8-20.

Listing 8-20. Editing the Connection String in the appsettings.json File

```
{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString":
"Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;Trusted_Connection=True;Multiple
ActiveResultSets=true"
    }
  }
}
```

Within the **Data** section of the configuration file, I have set the name of the connection string to **SportsStoreProducts**. The value of the **ConnectionString** item specifies that the LocalDB feature should be used for a database called **SportsStore**.

Tip Connection strings must be expressed as a single unbroken line, which is fine in the Visual Studio editor but doesn't fit on the printed page and explains the awkward formatting in Listing 8-20. When you define the connection string in your own project, make sure that the value of the `ConnectionString` item is on a single line.

Configuring the Application

The next steps are to read the connection string and to configure the application to use it to connect to the database. Another NuGet package is required to read the connection string from the `appsettings.json` file. Listing 8-21 shows the change to the `SportsStore.csproj` file.

Listing 8-21. Adding a Package in the SportsStore.csproj File of the SportsStore Project

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="Views\Shared\" />
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore"
      Version="2.0.0" />
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
    <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.1" />
    <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink"
      Version="1.1.0" />

    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
      Version="1.1.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
      Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
      Version="1.1.1" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json"
      Version="1.1.1" />
  </ItemGroup>
</Project>
```

CHAPTER 8 ? SportsStore

```
<DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
    Version="1.0.0" />
<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
    Version=" 1.0.0" />
</ItemGroup>

</Project>
```

This package allows configuration data to be read from JSON files, such as **appsettings.json**. A corresponding change is required in the **Startup** class to use the functionality provided by the new package to read the connection string from the configuration file and to set up EF Core, as shown in Listing 8-22.

Listing 8-22. Configuring the Application in the Startup.cs File

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

namespace SportsStore {

    public class Startup {
        IConfigurationRoot Configuration;

        public Startup(IHostingEnvironment env) {
            Configuration = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json").Build();
        }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProducts:ConnectionString"]));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app,
            IHostingEnvironment env, ILoggerFactory loggerFactory) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
        }
    }
}
```

```

        app.UseMvc(routes => {
            routes.MapRoute(
                name: "default",
                template: "{controller=Product}/{action=List}/{id?}");
        });
        SeedData.EnsurePopulated(app);
    }
}
}

```

The constructor that has been added to the `Startup` class loads the configuration settings in the `appsettings.json` file and makes them available through a property called `Configuration`. I explain how to read and access configuration data in Chapter 14.

Within the `ConfigureServices` method, I have added a sequence of method calls that sets up Entity Framework Core.

```

...
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration["Data:SportStoreProducts:ConnectionString"]));
...

```

The `AddDbContext` extension method sets up the services provided by Entity Framework Core for the database context class I created in Listing 8-17. As I explain in Chapter 14, many of the methods that are used in the `Startup` class allow services and middleware features to be configured using options arguments. The argument to the `AddDbContext` method is a lambda expression that receives an options object that configures the database for the context class. In this case, I configured the database with the `UseSqlServer` method and specified the connection string, which is obtained from the `Configuration` property.

The next change I made in the `Startup` class was to replace the fake repository with the real one, like this:

```

...
services.AddTransient<IProductRepository, EFProductRepository>();
...

```

The components in the application that use the `IProductRepository` interface, which is just the `Product` controller at the moment, will receive an `EFProductRepository` object when they are created, which will provide them with access to the data in the database. I explain how this works in detail in Chapter 18, but the effect is that the fake data will be seamlessly replaced by the real data in the database without having to change the `ProductController` class.

The final change in the `Startup` class is a call to the `SeedData.EnsurePopulated` method, which ensures that there is some sample data in the database and which I call from the `Configure` method in the `Startup` class. When the application starts, the `Startup.ConfigureServices` method is called before the `Startup.Configure` method, which

means that by the time the `SeedData.EnsurePopulated` method class is invoked, I can be sure that the Entity Framework Core services have been set up and configured.

Creating and Applying the Database Migration

Entity Framework Core is able to generate the schema for the database using the model classes through a feature called *migrations*. When you prepare a migration, EF Core creates a C# class that contains the SQL commands required to prepare the database. If you need to modify your model classes, then you can create a new migration that contains the SQL commands required to reflect the changes. In this way, you don't have to worry about manually writing and testing SQL commands and can just focus on the C# model classes in the application.

EF Core commands are performed using the Package Manager Console, which you can open through the Visual Studio Tools > NuGet Package Manager menu.

Run the following command in the Package Manager Console to create the migration class that will prepare the database for its first use:

```
Add-Migration Initial
```

When this command has finished, you will see a `Migrations` folder in the Visual Studio Solution Explorer window. This is where Entity Framework Core stores its migration classes. One of the file names will be a long number followed by `_Initial.cs`, and this is the class that will be used to create the initial schema for the database. If you examine the contents of this file, you can see how the `Product` model class has been used to create the schema.

Tip Restart Visual Studio if you can't run this command.

Run the following command to create the database and run the migration commands:

```
Update-Database
```

It can take a moment for the database to be created, but once the command has completed, you can see the effect of creating and using the database by starting the

application. When the browser requests the default URL for the application, the application configuration tells MVC that it needs to create a **Product** controller to handle the request. Creating a new **Product** controller means invoking the **ProductController** constructor, which requires an object that implements the **IProductRepository** interface, and the new configuration tells MVC that an **EFProductRepository** object should be created and used for this. The **EFProductRepository** taps into the EF Core functionality that loads relational data from SQL Server and converts it into **Product** objects. All of this is hidden from the **ProductController** class, which just receives an object that implements the **IProductRepository** interface and works with the data it provides. The result is that the browser window shows the sample data in the database, as illustrated by Figure 8-8.

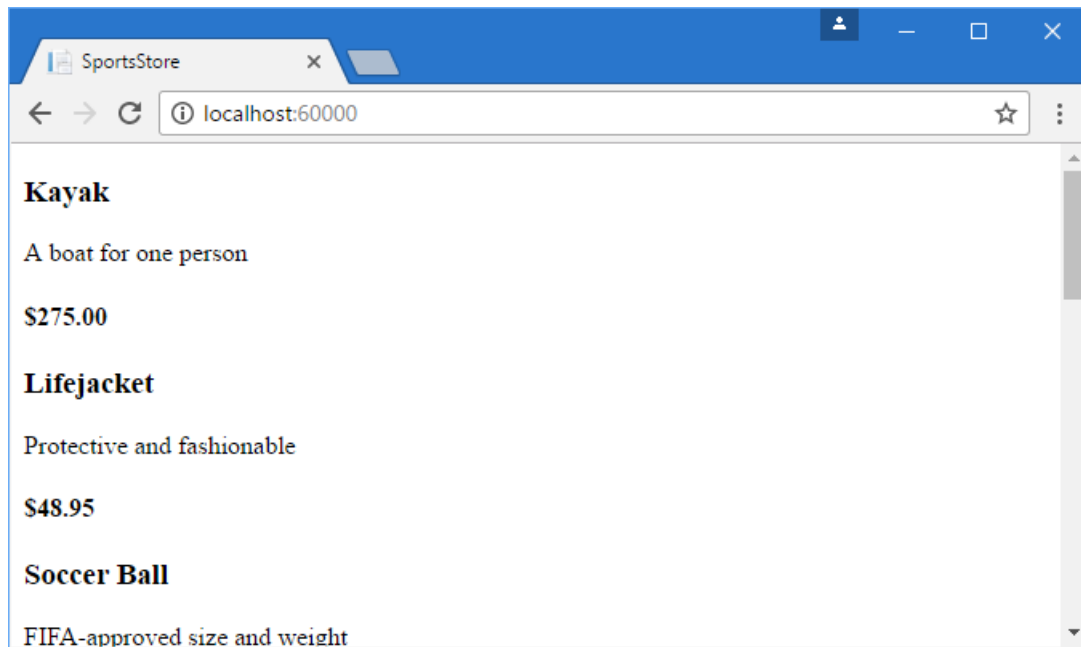


Figure 8-8. Using the database repository

This approach to getting Entity Framework Core 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 ASP.NET Core MVC. I am skipping over a lot of the detail in how EF Core operates and the huge number of configuration options that are available. I like Entity Framework Core 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 Entity Framework Core: <https://docs.microsoft.com/en-us/ef/core>.

Adding Pagination

You can see from Figure 8-8 that the `List.cshtml` view displays the products in the database on a single page. In this section, I will add support for pagination so that the view displays a smaller 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 8-23.

Listing 8-23. Adding Pagination Support to the List Action Method in the ProductController.cs File

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

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

        public ProductController(IProductRepository repo) {
            repository = repo;
        }

        public IActionResult List(int page = 1)
            => 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 replace this with a better mechanism later. 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 MVC 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 7 for details of how to set up unit tests. Here is the unit test I created for this purpose, in a class file called `ProductControllerTests.cs` that I added to the `SportsStore.Tests` project:

```
using System.Collections.Generic;
using System.Linq;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

    public class ProductControllerTests {

        [Fact]
        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 =
                controller.List(2).ViewData.Model as IEnumerable<Product>;

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

```

    }
  }
}

```

It is a little awkward to get the data returned from the action method. The result is a `ViewResult` object, and I have to cast the value of its `ViewData.Model` property to the expected data type. I explain the different result types that can be returned by action methods and how to work with them in Chapter 17.

Displaying Page Links

If you run the application, you will see that there are now 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:60000/?page=2
```

You will need to change the port part of the URL to match whatever port has been assigned to your project. Using these query strings, you can navigate through the catalog of products.

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 each list of products so that customers can navigate between pages. To do this, I am going to implement a *tag helper*, which generates the HTML markup for the links I require.

Adding the View Model

To support the tag 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 class, which is used specifically to pass data between a controller and a view. I created a `Models/ViewModels` folder in the `SportsStore` project and added to it a class file called `PagingInfo.cs` defined in Listing 8-24.

Listing 8-24. The Contents of the `PagingInfo.cs` File in the `Models/ViewModels` Folder

```
using System;
```


```
namespace SportsStore.Models.ViewModels {

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

        public int TotalPages =>
            (int)Math.Ceiling((decimal)TotalItems / ItemsPerPage);
    }
}
```

Adding the Tag Helper Class

Now that I have a view model, I can create a tag helper class. I created the **Infrastructure** folder in the **SportsStore** project and added to it a class file called **PageLinkTagHelper.cs**, which I used to define the class shown in Listing 8-25. Tag helpers are a big part of ASP.NET Core MVC, and I explain how they work and how to create them in Chapters 23–25.

 **Tip** The **Infrastructure** folder is where I put classes that deliver the plumbing for an application but that are not related to the application’s domain.

Listing 8-25. The Contents of the PageLinkTagHelper.cs File in the Infrastructure Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;

namespace SportsStore.Infrastructure {

    [HtmlTargetElement("div", Attributes = "page-model")]
    public class PageLinkTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public PageLinkTagHelper(IUrlHelperFactory helperFactory) {
            urlHelperFactory = helperFactory;
        }

        [ViewContext]
        [HtmlAttributeNotBound]
    }
```

```

    public ViewContext ViewContext { get; set; }

    public PagingInfo PageModel { get; set; }

    public string PageAction { get; set; }

    public override void Process(TagHelperContext context,
        TagHelperOutput output) {
        IHttpHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
        TagBuilder result = new TagBuilder("div");
        for (int i = 1; i <= PageModel.TotalPages; i++) {
            TagBuilder tag = new TagBuilder("a");
            tag.Attributes["href"] = urlHelper.Action(PageAction,
                new { page = i });
            tag.InnerHtml.Append(i.ToString());
            result.InnerHtml.AppendHtml(tag);
        }
        output.Content.AppendHtml(result.InnerHtml);
    }
}

```

This tag helper populates a **div** element with **a** elements that correspond to pages of products. I am not going to go into detail about tag helpers now; it is enough to know that they are one of the most useful ways that you can introduce C# logic into your views. The code for a tag helper can look tortured because C# and HTML don't mix easily. But using tag helpers is preferable to including blocks of C# code in a view because a tag helper can be easily unit tested.

Most MVC components, such as controllers and views, are discovered automatically, but tag helpers have to be registered. In Listing 8-26, I have added a statement to the **_ViewImports.cshtml** file in the **Views** folder that tells MVC to look for tag helper classes in the **SportsStore.Infrastructure** namespace. I also added an **@using** expression so that I can refer to the view model classes in views without having to qualify their names with the namespace.

Listing 8-26. Registering a Tag Helper in the _ViewImports.cshtml File

```

@using SportsStore.Models
@using SportsStore.Models.ViewModels
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper SportsStore.Infrastructure.*, SportsStore

```

UNIT TEST: CREATING PAGE LINKS

To test the `PageLinkTagHelper` tag helper class, I call the `Process` method with test data and provide a `TagHelperOutput` object that I inspect to see the HTML that is generated, as follows, which I defined in a new `PageLinkTagHelperTests.cs` file in the `SportsStore.Tests` project:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Moq;
using SportsStore.Infrastructure;
using SportsStore.Models.ViewModels;
using Xunit;

namespace SportsStore.Tests {

    public class PageLinkTagHelperTests {

        [Fact]
        public void Can_Generate_Page_Links() {
            // Arrange
            var urlHelper = new Mock<IUrlHelper>();
            urlHelper.SetupSequence(x => x.Action(It.IsAny<UrlActionContext>()))
                .Returns("Test/Page1")
                .Returns("Test/Page2")
                .Returns("Test/Page3");

            var urlHelperFactory = new Mock<IUrlHelperFactory>();
            urlHelperFactory.Setup(f =>
                f.GetUrlHelper(It.IsAny<ActionContext>()))
                .Returns(urlHelper.Object);

            PageLinkTagHelper helper =
                new PageLinkTagHelper(urlHelperFactory.Object) {
                    PageModel = new PagingInfo {
                        CurrentPage = 2,
                        TotalItems = 28,
                        ItemsPerPage = 10
                    },
                    PageAction = "Test"
                };
        }
    }
}
```

```

TagHelperContext ctx = new TagHelperContext(
    new TagHelperAttributeList(),
    new Dictionary<object, object>(), "");

var content = new Mock<TagHelperContent>();
TagHelperOutput output = new TagHelperOutput("div",
    new TagHelperAttributeList(),
    (cache, encoder) => Task.FromResult(content.Object));

// Act
helper.Process(ctx, output);

// Assert
Assert.Equal(@"<a href=""Test/Page1"">1</a>"
    + @"<a href=""Test/Page2"">2</a>"
    + @"<a href=""Test/Page3"">3</a>",
    output.Content.GetContent());
    }
}
}

```

The complexity in this test is in creating the objects that are required to create and use a tag helper. Tag helpers use **IUrlHelperFactory** objects to generate URLs that target different parts of the application, and I have used Moq to create an implementation of this interface and the related **IUrlHelper** interface that provides test data.

The core part of the test verifies the tag helper 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 uses 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 several lines because the width of a printed page is narrow. I have not added a newline character; if I did, the test would fail.

Adding the View Model Data

I am not quite ready to use the tag helper because 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/ViewModels** folder of the **SportsStore** project. Listing 8-27 shows the contents of the new file.

Listing 8-27. The Contents of the ProductsListViewModel.cs File in the Models/ViewModels Folder

```

using System.Collections.Generic;
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

    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 8-28.

Listing 8-28. Updating the List Method in the ProductController.cs File

```

using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

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

        public ProductController(IProductRepository repo) {
            repository = repo;
        }

        public IActionResult List(int page = 1)
            => View(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()
                }
            });
    }
}

```

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 `ProductControllerTests` class in the test project to make sure:

```
...
[Fact]
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) { PageSize = 3 };

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

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

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:

```
...
```



```
[Fact]
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 =
        controller.List(2).ViewData.Model as ProductsListViewModel;

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

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 8-29, to deal with the new view model type.

Listing 8-29. Updating the List.cshtml File

```
@model ProductsListViewModel

@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 add an HTML element that the tag helper will process to create the page links, as shown in Listing 8-30.

Listing 8-30. Adding the Pagination Links in the List.cshtml File

```
@model ProductsListViewModel

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

<div page-model="@Model.PagingInfo" page-action="List"></div>
```

If you run the application, you will see the new page links, as illustrated in Figure 8-9. 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 allow for exploration of the products for sale. When Razor finds the `page-model` attribute on the `div` element, it asks the `PageLinkTagHelper` class to transform the element, which produces the set of links shown in the figure.

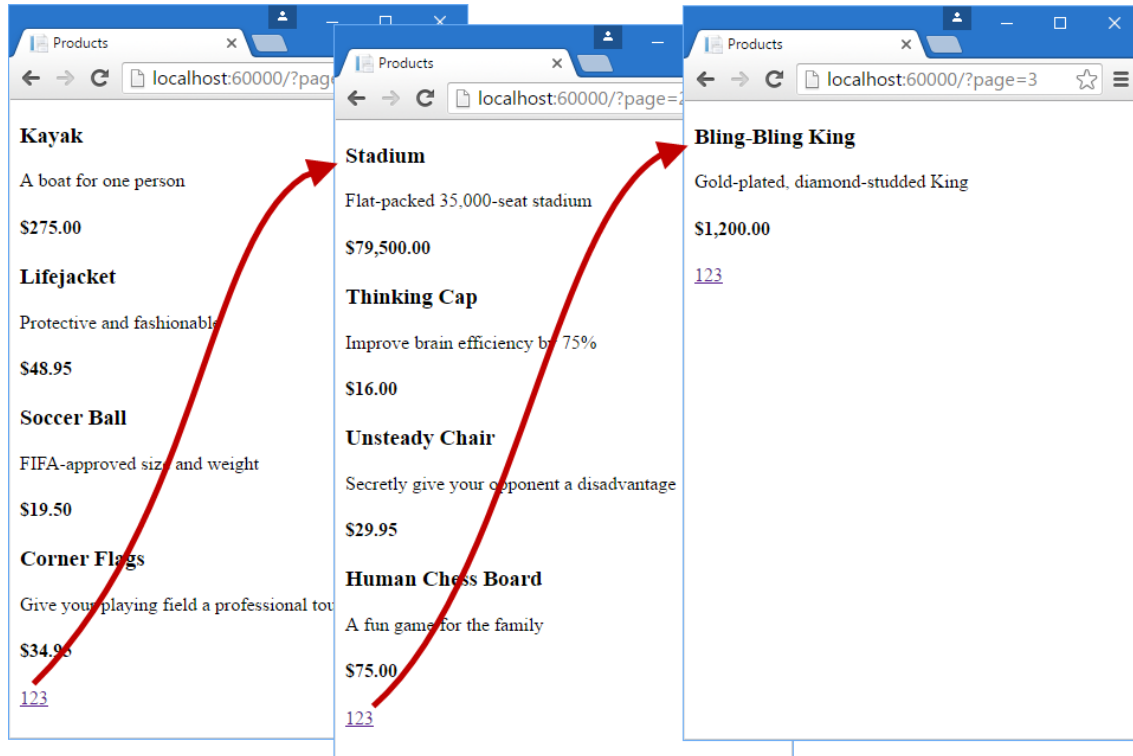


Figure 8-9. Displaying page navigation links

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 simple paginated product 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 profoundly 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, which is 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 an application because it uses the ASP.NET *routing* feature, which is responsible for processing URLs to figure out what part of the application they target. All I need to do is add a new route when registering the MVC middleware in the `Configure` method of the `Startup` class, as shown in Listing 8-31.

Listing 8-31. Adding a New Route in the Startup.cs File

```
...
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env, ILoggerFactory loggerFactory) {
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseMvc(routes => {

        routes.MapRoute(
            name: "pagination",
            template: "Products/Page{page}",
            defaults: new { Controller = "Product", action = "List" });
    });
}
```

```
        routes.MapRoute(
            name: "default",
            template: "{controller=Product}/{action=List}/{id?}");
    });
    SeedData.EnsurePopulated(app);
}
...
```

It is important that you add this route before the **Default** one that is already in the method. 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. MVC and the routing function are tightly integrated, so the application automatically reflects a change like this in the URLs used by the application, including those generated by tag helpers like the one I use to generate the page navigation 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 click a pagination link, you will see the new URL scheme in action, as illustrated in Figure 8-10.

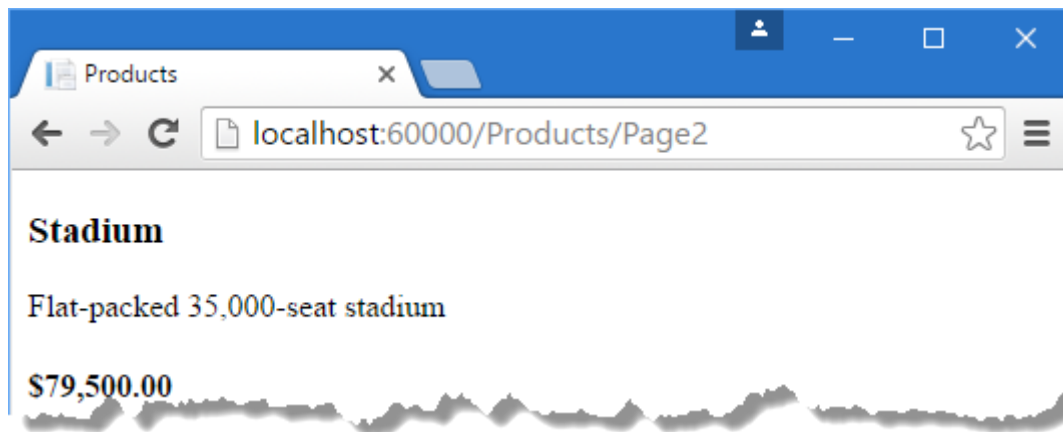


Figure 8-10. The new URL scheme displayed in the browser

Styling the Content

I have built a great deal of infrastructure and the basic features of the application are starting to come together, but I have not paid any attention to 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 8-11.

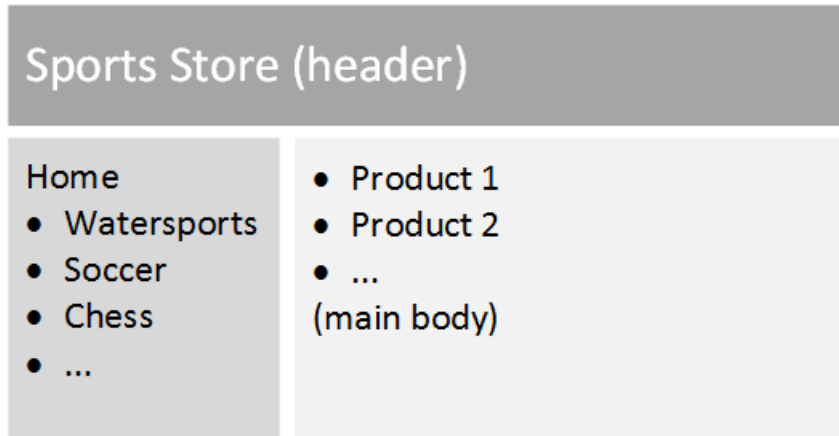


Figure 8-11. The design goal for the SportsStore application

Installing the Bootstrap Package

I am going to use the Bootstrap package to provide the CSS styles I will apply to the application. I will rely on the Visual Studio support for Bower to install the Bootstrap package for me, so I selected the Bower Configuration File item template from the Web > General category of the Add New Item dialog to create a file called **bower.json** in the SportsStore project, as demonstrated in Chapter 6. I then added the Bootstrap package to the **dependencies** section of the file that was created, as shown in Listing 8-32.

Listing 8-32. Adding Bootstrap to the *bower.json* File in the SportsStore Project

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6"
  }
}
```

When the changes to the **bower.json** file are saved, Visual Studio uses Bower to download the Bootstrap package into the **wwwroot/lib/bootstrap** folder. Bootstrap depends on the jQuery package, and this will be automatically added to the project as well.

Applying Bootstrap Styles to the Layout

In Chapter 5, I explained how Razor layouts work, how they are used, and how they incorporate layouts. The view start file that I added at the start of the chapter specified that a file called `_Layout.cshtml` should be used as the default layout, and that is where the initial Bootstrap styling will be applied, as shown in Listing 8-33.

Listing 8-33. Applying Bootstrap CSS to the `_Layout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>SportsStore</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>
```

The `link` element in this listing has an `asp-href-include` attribute, which represents an example of a built-in tag helper class. In this case, the tag helper looks at the value of the attribute and generates `link` elements for all the files that match the specified path, which can include wildcards. This is a useful feature to ensure that you can add and remove files from the `wwwroot` folder structure without breaking the application, but, as I explain in Chapter 25, caution is required to make sure that the wildcards you specify match the files you expect.

Adding the Bootstrap CSS stylesheet to the layout means that I can use the styles it defines in any of the views that rely on the layout. In Listing 8-34, you can see the styling I applied to the `List.cshtml` file.

Listing 8-34. Styling Content in the `List.cshtml` File

```
@model ProductsListViewModel
```

```

@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 page-model="@Model.PagingInfo" page-action="List" page-classes-enabled="true"
    page-class="btn" page-class-normal="btn-default"
    page-class-selected="btn-primary" class="btn-group pull-right">
</div>

```

I need to style the buttons that are generated by the `PageLinkTagHelper` class, but I don't want to hardwire the Bootstrap classes into the C# code because it makes it harder to reuse the tag helper elsewhere in the application or change the appearance of the buttons. Instead, I have defined custom attributes on the `div` element that specify the classes that I require, and these correspond to properties I added to the tag helper class, which are then used to style the `a` elements that are produced, as shown in Listing 8-35.

Listing 8-35. Adding Classes to Generated Elements in the `PageLinkTagHelper.cs` File

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;

namespace SportsStore.Infrastructure {

    [HtmlTargetElement("div", Attributes = "page-model")]
    public class PageLinkTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public PageLinkTagHelper(IUrlHelperFactory helperFactory) {
            urlHelperFactory = helperFactory;
        }

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext ViewContext { get; set; }
    }
}

```



```

public PagingInfo PageModel { get; set; }

public string PageAction { get; set; }

public bool PageClassesEnabled { get; set; } = false;
public string PageClass { get; set; }
public string PageClassNormal { get; set; }
public string PageClassSelected { get; set; }

public override void Process(TagHelperContext context,
    TagHelperOutput output) {
    IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
    TagBuilder result = new TagBuilder("div");
    for (int i = 1; i <= PageModel.TotalPages; i++) {
        TagBuilder tag = new TagBuilder("a");
        tag.Attributes["href"] = urlHelper.Action(PageAction,
            new { page = i });
        if (PageClassesEnabled) {
            tag.AddCssClass(PageClass);
            tag.AddCssClass(i == PageModel.CurrentPage
                ? PageClassSelected : PageClassNormal);
        }
        tag.InnerHtml.Append(i.ToString());
        result.InnerHtml.AppendHtml(tag);
    }
    output.Content.AppendHtml(result.InnerHtml);
}
}

```

The values of the attributes are automatically used to set the tag helper property values, with the mapping between the HTML attribute name format (`page-class-normal`) and the C# property name format (`PageClassNormal`) taken into account. This allows tag helpers to respond differently based on the attributes of an HTML element, creating a more flexible way to generate content in an MVC application.

If you run the application, you will see that the appearance of the application has been improved—at least a little, anyway—as illustrated by Figure 8-12.

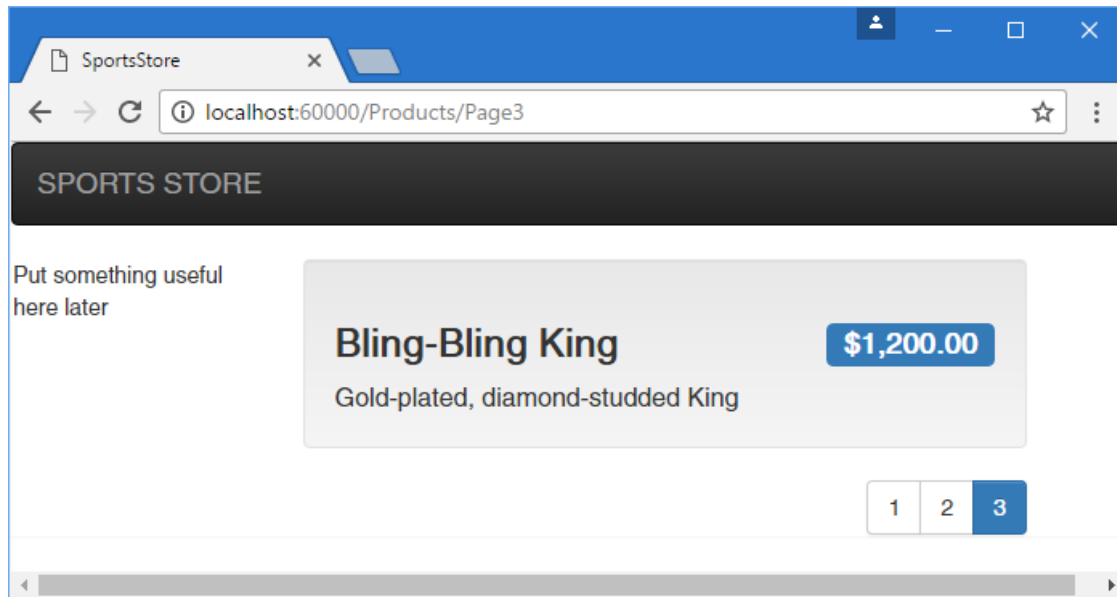


Figure 8-12. The design-enhanced SportsStore application

Creating a Partial View

As a finishing flourish 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. I describe partial views in detail in Chapter 21, and they help reduce duplication when you need the same content to appear in different places in an application. Rather than copy and paste the same Razor markup into multiple views, you can define it once in a partial view. To create the partial view, I added a Razor view file called `ProductSummary.cshtml` to the `Views/Shared` folder and added the markup shown in Listing 8-36.

Listing 8-36. The Contents of the `ProductSummary.cshtml` File in the `Views/Shared` Folder

```
@model Product

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

```

    </h3>
    <span class="lead">@Model.Description</span>
</div>

```

Now I need to update the `List.cshtml` file in the `Views/Products` folder so that it uses the partial view, as shown in Listing 8-37.

Listing 8-37. Using a Partial View in the List.cshtml File

```

@model ProductsListViewModel

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

<div page-model="@Model.PagingInfo" page-action="List" page-classes-enabled="true"
    page-class="btn" page-class-normal="btn-default"
    page-class-selected="btn-primary" class="btn-group pull-right">
</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, with arguments for the name of the view and the view model object. Switching to a partial view like this is good practice because it allows the same markup to be inserted into any view that needs to display a summary of a product. As Figure 8-13 shows, adding the partial view doesn't change the appearance of the application; it just changes where Razor finds the content that is used to generate the response sent to the browser.

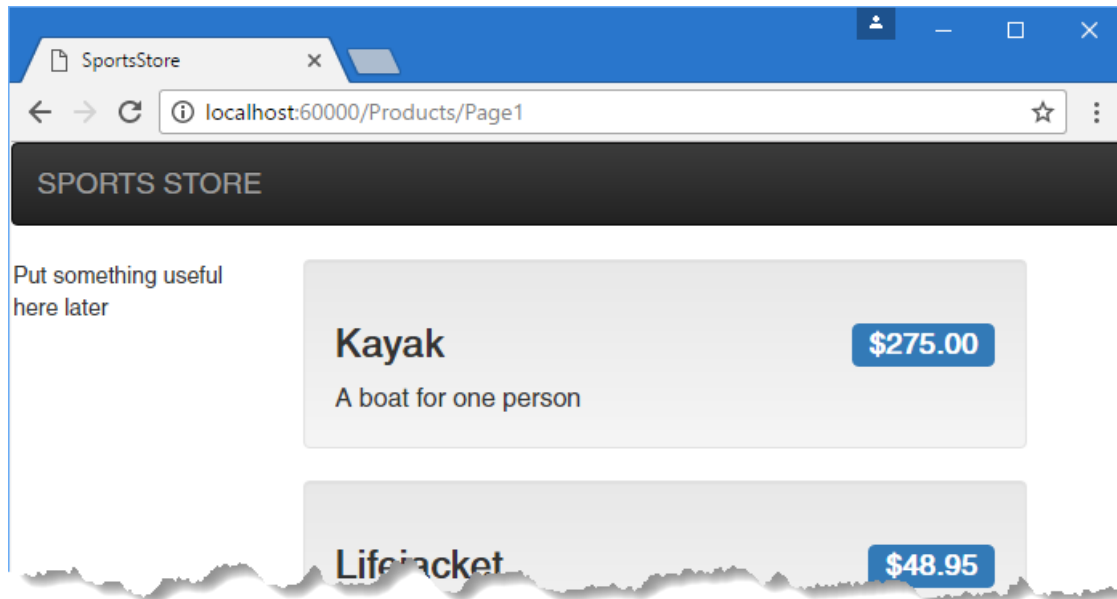


Figure 8-13. Applying a partial view

Summary

In this chapter, I built 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 Core. There is a single controller, `ProductController`, that can produce paginated lists of products, and I have set up 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 the customer-facing features: navigation by category, a shopping cart, and the start of a checkout process.