

ASP.NET Core MVC with Entity Framework Tutorial

Karthikeyan Umapathy
UNF School of Computing

Conditions and Terms of Use

This tutorial is intended for academic training purposes only. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or distributing all or any portion of the content included in this tutorial is strictly prohibited. No part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission from the instructor of the course. Complying with the conditions and terms of use stated here is the responsibility of the user.

Prerequisites

This tutorial teaches basis of building an ASP.NET Core MVC web app using Visual Studio Professional 2017. This tutorial is designed for beginners who want to build ASP.NET Core web application step by step. Basic knowledge of C#, HTML, CSS, Visual Studio, and Object Oriented Programming are required to take full advantage of this tutorial.

Resources Used for Preparing Tutorial

This tutorial is prepared based on materials obtained from Microsoft, MVC book by Adam Freeman, and other online training resources.

1. Microsoft MVC Training Package received to be strictly used for academic and classroom learning purposes.
2. Pro ASP.NET Core MVC by Adam Freeman (<http://www.apress.com/us/book/9781484203989>)
3. Microsoft Docs (<https://docs.microsoft.com/en-us/aspnet/core/>)

Table of Contents

I	Exercise 1: Creating SportsStore: A Realistic Online Application	4
I.1	Task 1: Creating the MVC Project.....	4
I.2	Task 2: Adding the NuGet Packages.....	6
I.3	Task 3: Creating the Folder Structure	9
I.4	Task 4: Configuring the Application.....	9
I.5	Task 5: Importing Razor Views	11
I.6	Task 6: Checking and Running the Application	12
I.7	Task 7: Creating Product Domain Model.....	13
I.8	Task 8: Creating a Product Repository Interface	14
I.9	Task 9: Creating a Fake Repository	15
I.10	Task 10: Registering the Repository Service	15
I.11	Task 11: Creating Controller for Displaying a List of Products.....	16
I.12	Task 12: Adding and Configuring the Layout of the View	18
I.13	Task 13: Adding and Configuring ViewStart	19
I.14	Task 14: Adding View to Display Product Listing.....	20
I.15	Task 15: Setting the Default Route	21
I.16	Task 16: Running the Application.....	22
I.17	Preparing a Database.....	23
I.18	Task 17: Installing Entity Framework Core.....	23
I.19	Task 18: Creating the Database Classes.....	24
I.20	Task 19: Creating the Repository Class	25
I.21	Task 20: Obtaining the Database Connection String.....	25
I.22	Task 21: Defining the Connection String.....	29
I.23	Task 22: Configuring the Application	31
I.24	Task 23: Creating and Applying the Database Migration	33
I.25	Task 24: Running the Application.....	39
I.26	Task 25: Adding Pagination	40
I.27	Task 26: Adding the View Model.....	43
I.28	Task 27: Adding the Tag Helper Class	45
I.29	Task 28: Adding the View Model Data.....	48
I.30	Task 29: Displaying the Page Links.....	50
I.31	Task 30: Improving the URLs	51
I.32	Styling the Content.....	55
I.33	Task 31: Installing the Bootstrap Package.....	55

I.34	Task 32: Applying Bootstrap Styles to the Layout.....	59
I.35	Task 33: Creating a Partial View.....	62
I.36	Adding Navigation Controls.....	64
I.37	Task 34: Filtering the Product List.....	64
I.38	Task 35: Refining the URL Scheme.....	66
I.39	Task 36: Building a Category Navigation Menu.....	71
I.40	Task 37: Building the Shopping Cart.....	81
I.41	Task 38: Enabling Sessions.....	84
I.42	Task 39: Implementing the Cart Controller.....	86
I.43	Task 40: Displaying the Contents of the Cart.....	89
I.44	Refining the Cart Model with a Service.....	93
I.45	Task 41: Creating a Storage-Aware Cart Class.....	93
I.46	Task 42: Removing Items from the Cart.....	97
I.47	Task 43: Adding the Cart Summary Widget.....	99
I.48	Submitting Orders.....	102
I.49	Task 44: Creating the Model Class.....	102
I.50	Task 45: Adding the Checkout Process.....	104
I.51	Task 46: Implementing Order Processing.....	108
I.52	Task 47: Completing the Order Controller.....	111
I.53	Task 48: Displaying Validation Errors.....	112
I.54	Task 49: Displaying a Summary Page.....	114
I.55	Task 50: Managing Orders.....	115
I.56	Task 51: Adding the Actions and View.....	116
I.57	Task 52: Adding Catalog Management.....	120
I.58	Task 53: Editing Products.....	122
I.59	Task 54: Updating the Product Repository.....	125
I.60	Task 55: Handling Edit POST Requests.....	126
I.61	Task 56: Adding Model Validation.....	129
I.62	Task 57: Enabling Client-Side Validation.....	132
I.63	Task 58: Creating New Products.....	134
I.64	Task 59: Deleting Products.....	138
I.65	Securing the Administration Features.....	141
I.66	Task 60: Create Application User class inheriting IdentityUser base class.....	141
I.67	Task 61: Modify ApplicationDbContext class.....	141
I.68	Task 62: Creating and Applying the Database Migration.....	144
I.69	Task 63: Creating the Account Controller.....	147

I.70	Task 64: Creating Login View	151
I.71	Task 65: Creating User Accounts	152
I.72	Task 66: Listing User Accounts	153
I.73	Task 67: Create a User Account.....	154
I.74	Task 68: Testing the Security Policy	155
I.75	Summary.....	156

1 Exercise 1: Creating SportsStore: A Realistic Online Application

In the previous tutorial, we built quick and simple MVC application. Now it is time to build a simple but realistic e-business application. We will create an online application called SportsStore following classic approach taken by online stores everywhere. We will create an online product catalog that customers can browse by category, a shopping cart where users can add and remove products, and a checkout where customers can enter their shipping details. We will also create an administration area that includes create, read, update, and delete (CRUD) facilities for managing the catalog, and we will protect it so that only logged-in administrators can make changes.

The goal of this exercise is to give you a sense of what real MVC development is like by creating as realistic an example as possible. We want to focus on the ASP.NET Core MVC, of course, so we 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 we build up the levels of infrastructure needed, but the initial investment in an MVC application pays dividends, resulting in maintainable, extensible, and well-structured code.

1.1 Task 1: Creating the MVC Project

Follow the same basic approach that we used in the previous tutorial, but this time we create an empty project. **Note, for this exercise, for step 5, we will choose Empty as the template option.**

1. Open Visual Studio.
2. Create a new ASP.NET Core application project by going to File > New Project.
3. Under templates, go to Visual C# > Web, and then choose ASP.NET Core Web Application (.NET Core). See Figure 1.
4. Name the project SportsStore, and change the location as per your own preferences. Solution should be set to Create new solution, and leave the check box selected for Create directory for solution. Click OK and you will see another dialog box, which asks you to set the initial content for the project.
5. Choose Empty under ASP.NET Core Templates as shown in Figure 2. Use the “Change Authentication” button to select “No authentication”. Ensure that the Host in the Cloud option is unchecked.
6. Click OK to create the project.
7. Visual Studio will take a few seconds to setup a project.

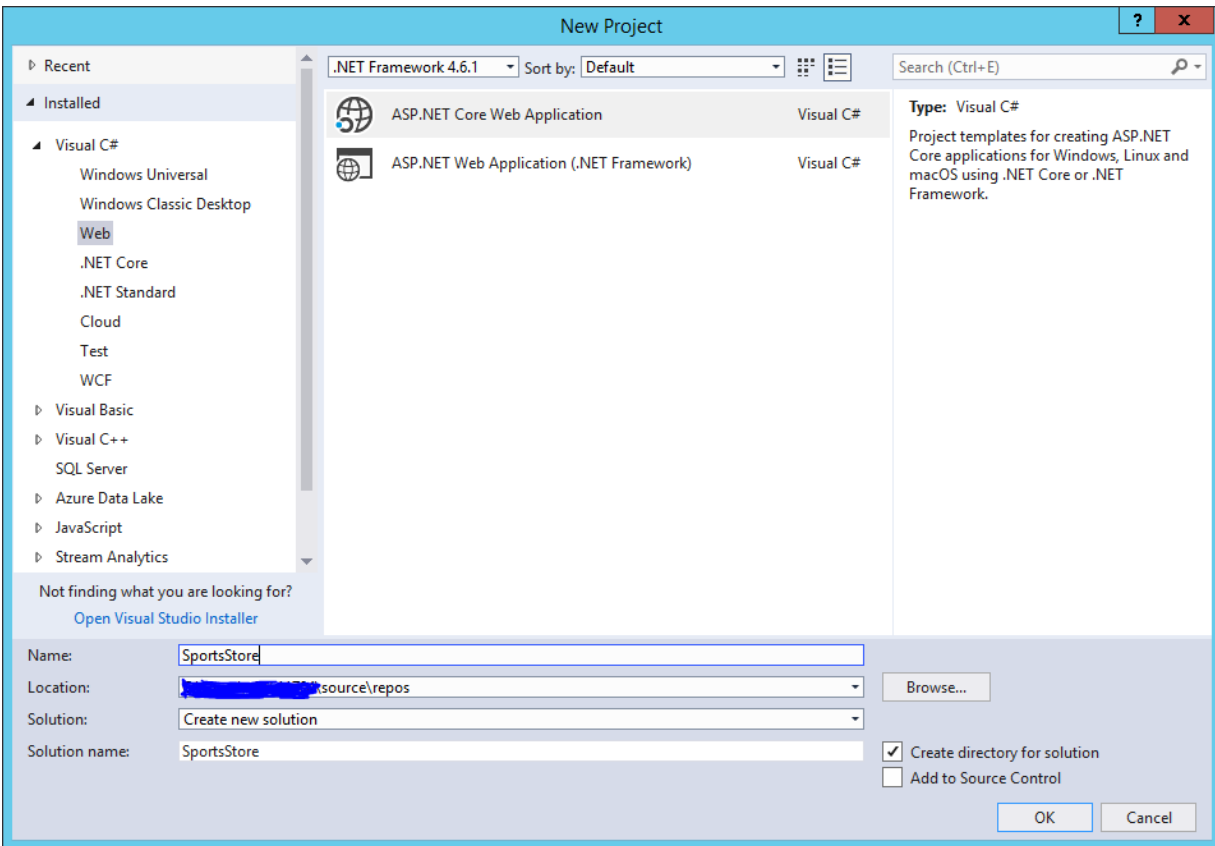


Figure 1. Creating ASP.NET Core Web Application Project

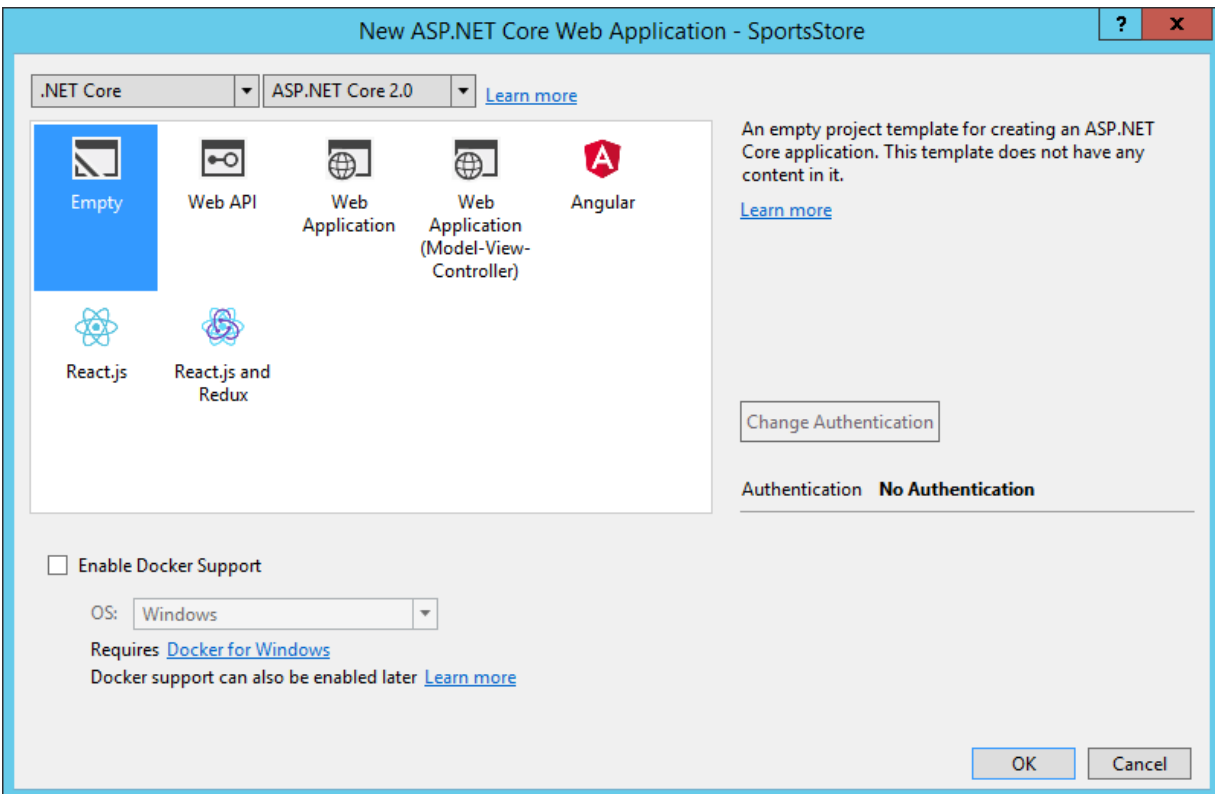


Figure 2. Selecting Empty Template for the Web Application Project

1.2 Task 2: Adding the NuGet Packages

Visual Studio provides a graphical tool for managing the .NET packages that are included in a project. To open the tool, select Manage NuGet Packages for Solution from the Tools ► NuGet Package Manager menu. The NuGet tool opens and displays a list of the packages that are already installed, as shown in Figure 3. The Installed tab provides a summary of the packages that are already installed in the project. The Browse tab can be used to locate and install new packages and the Updates tab can be used to list packages for which more recent versions have been released.

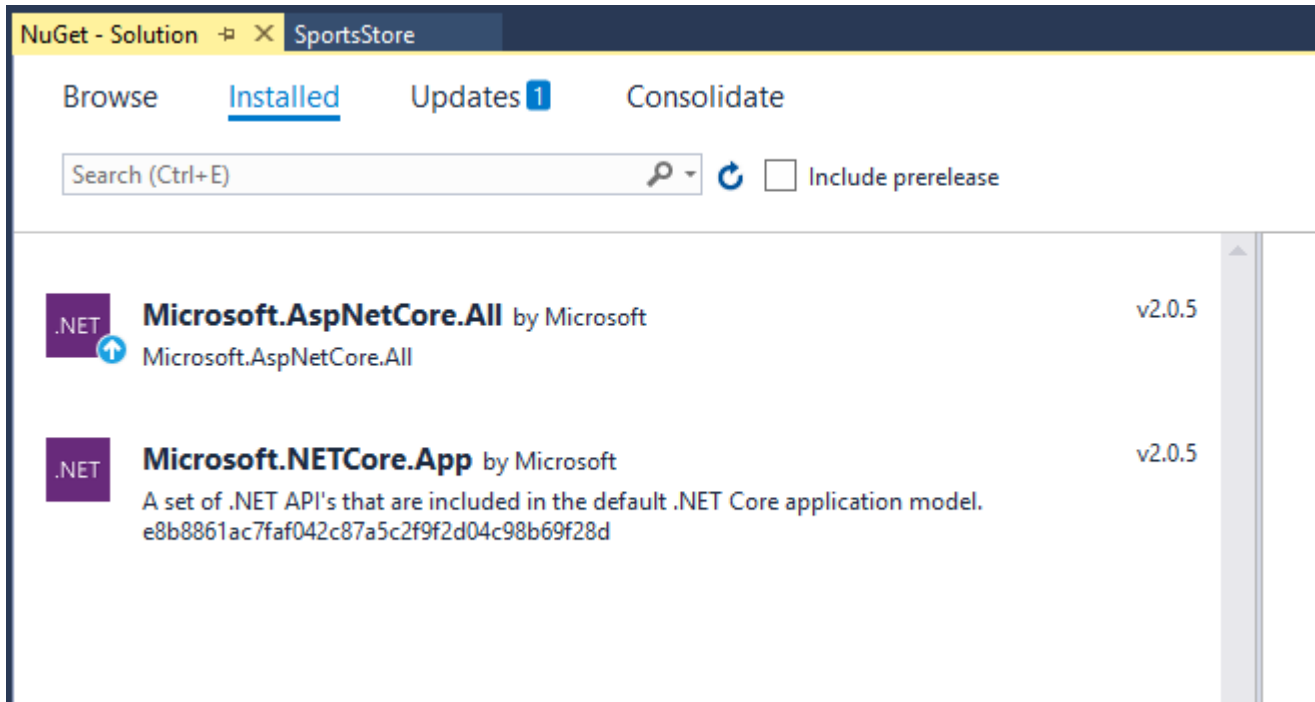


Figure 3. Manage NuGet Packages for Solution Window

The NuGet tool manages the contents of the dependencies in a csproj file, which is created by Visual Studio when a new project is set up, even when using the Empty template. 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, as shown in Figure 4. The Empty project template installs the basic ASP.NET Core features but requires additional packages to provide functionality required for MVC applications. Listing I shows the additions we need to make to the SportsStore.csproj file to add the packages needed to get started with the SportsStore application.

Listing I. Adding NuGet Packages in the SportsStore.csproj File

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
```



```

<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />
  <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="2.0.1" />
</ItemGroup>

<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
Version="2.0.0" />
</ItemGroup>

</Project>

```

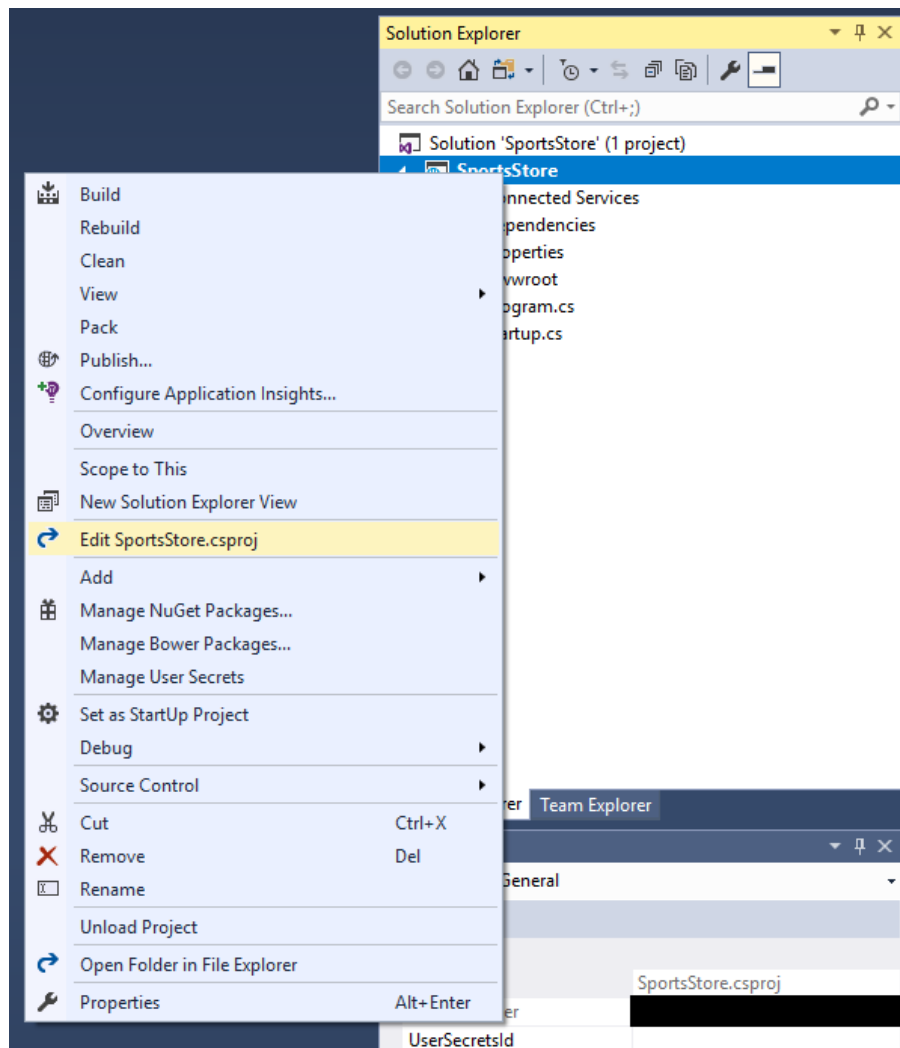


Figure 4. Using the NuGet package manager

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

Table I. The Essential NuGet Packages Installed

Name	Description
Microsoft.AspNetCore.All	This package contains entire ASP.NET Core dependencies including ASP.NET Core MVC which provides access to essential features such as controllers and Razor views as well as ASP.NET Core StaticFiles package which 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.

If you inspect the csproj file, you will note that each package is specified with its name and the version number that is required. Visual Studio monitors the contents of the csproj file, which means that you can add or remove packages by editing the file directly, which is what we will do because it helps ensure that you will get the expected results if you are following along.

When you add a NuGet package to a project, it is automatically installed along with any packages it depends on. You can explore the NuGet packages and their dependencies by opening the Dependencies item in the Solution Explorer, which contains an entry for each NuGet package in the csproj file, as shown in Figure 5. As the figure shows, when we added the Microsoft.VisualStudio.Web.BrowserLink package in Listing 1, NuGet downloaded and installed that package and many others that are required for MVC development.

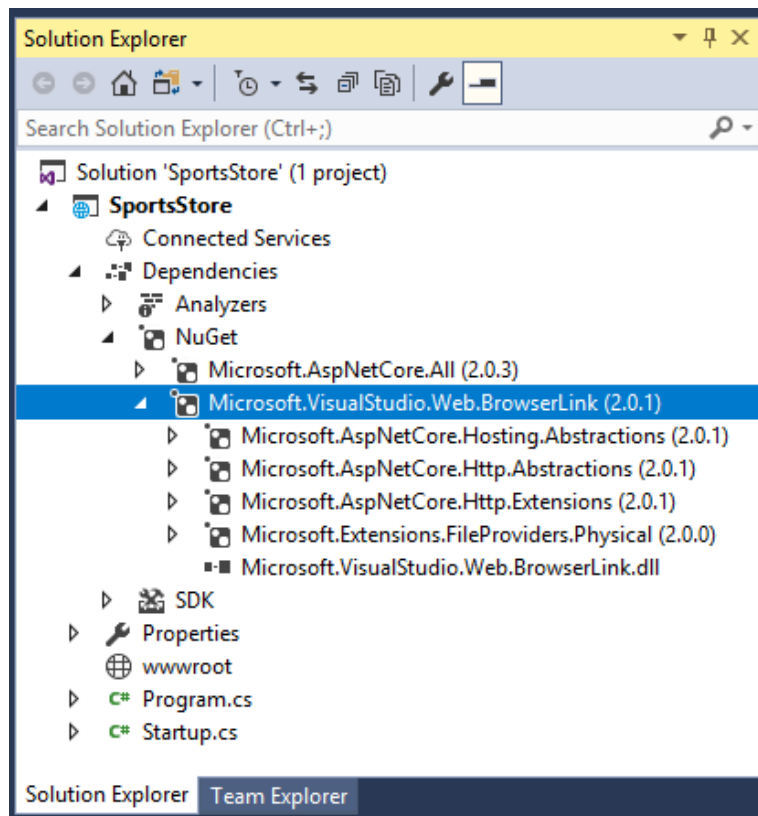


Figure 5. The Dependencies section of the Solution Explorer

1.3 Task 3: 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, right-click the SportsStore project item in the Solution Explorer, 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.

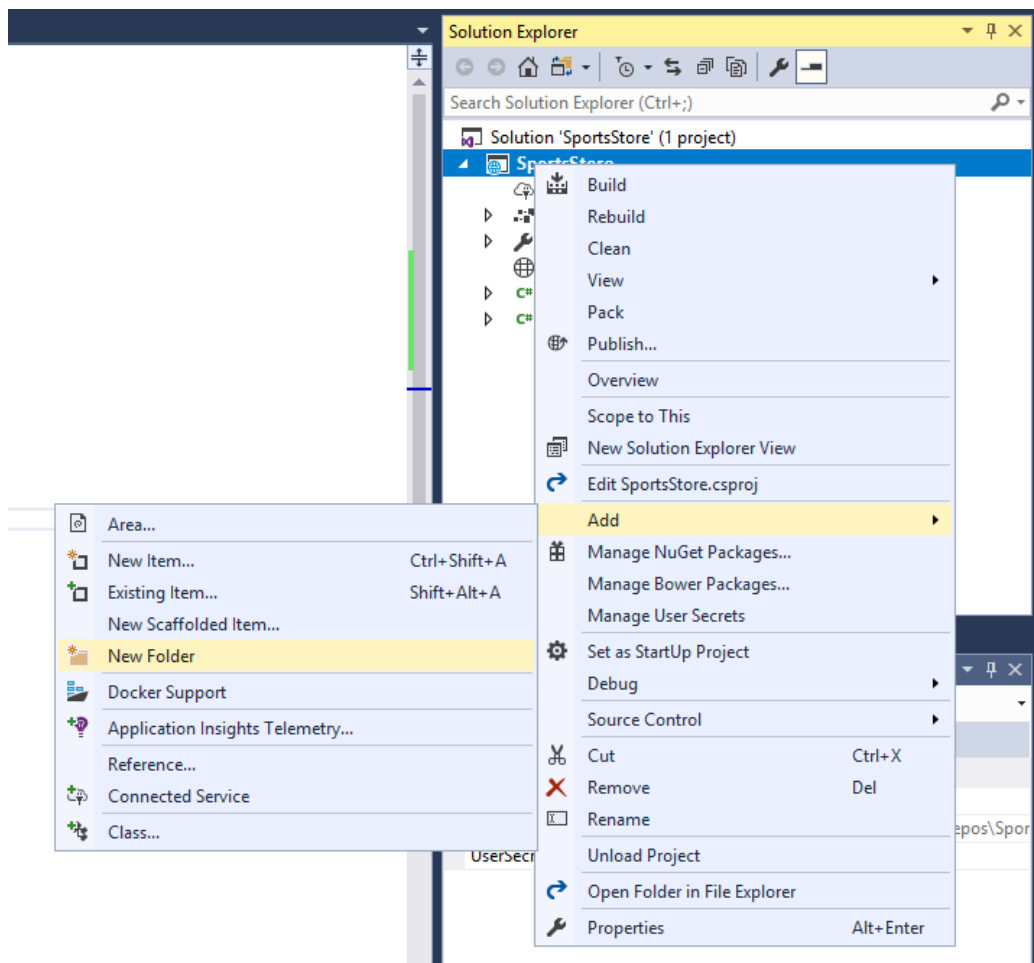


Figure 6. The Adding Models, Controllers, and Views Folders

1.4 Task 4: Configuring the Application

An ASP.NET Core MVC application relies on several configuration files. First, having installed the NuGet packages, we need to edit the Startup class to tell ASP.NET to use them, as shown in Listing 2. To edit the Startup class, double click on the Startup.cs file available in the solution explorer.

Listing 2. Enabling Features in the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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
    {
        // This method gets called by the runtime. Use this method to add services to the
        // container.
        // For more information on how to configure your application, visit
        // https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        // pipeline.
        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. The AddMvc method that we 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 we call in the Configure method is an extension method that sets up an HTTP request processor, as described in Table 2.

Table 2. 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. Note: It should not be enabled in deployed applications.
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.

1.5 Task 5: Importing Razor Views

We 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 Figures 7 and 8. Click the Add button to create the `_ViewImports.cshtml` file and set the contents of the new file to match Listing 3. The purpose of the `_ViewImports.cshtml` file is to provide a mechanism to make directives available to Razor pages globally so that you don't have to add them to pages individually. You can learn more about ViewImports file in relation to Razor pages at <https://www.learnrazorpages.com/razor-pages/files/viewimports>.

The `@using` statement will allow us 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 we use later to create HTML elements that reflect the configuration of the `SportsStore` application.

When you save the `_ViewImports` file, you will note red line underneath `Models` as shown in Figure 9. For now, you can ignore this error. This error is appearing as we have not created any models.

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

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

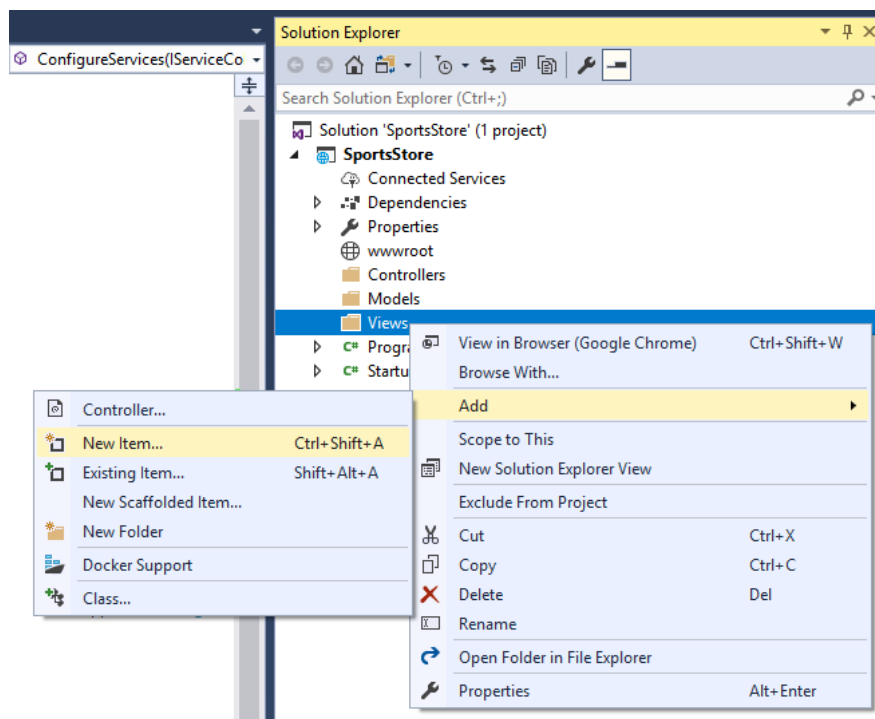


Figure 7. Adding new item to Views

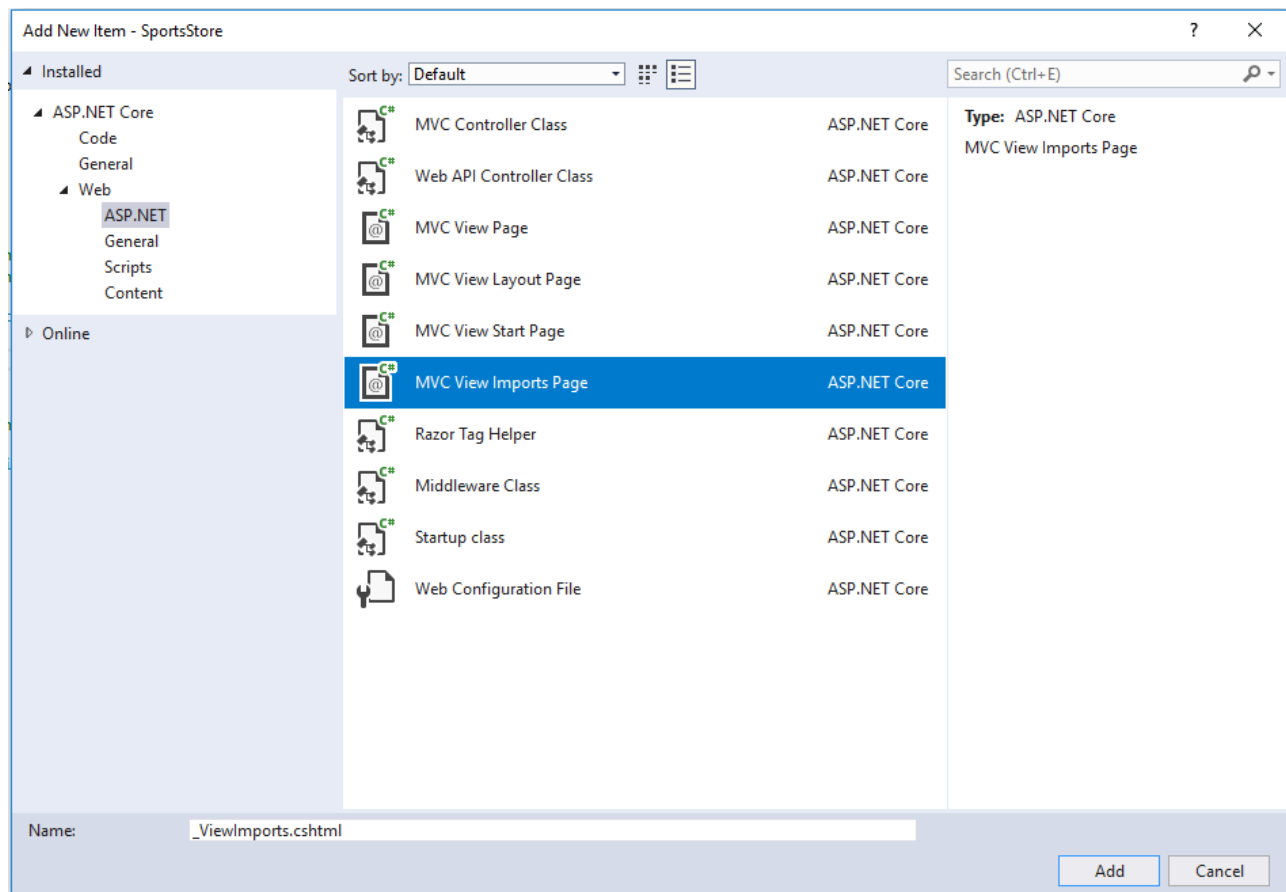


Figure 8. Creating the view imports file

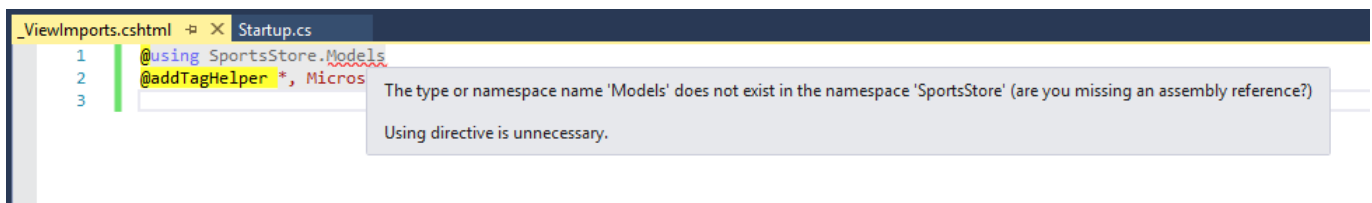


Figure 9. Models namespace error

1.6 Task 6: Checking and Running the Application

If you select SportsStore project in the solution explorer and then select Start Debugging from the Debug menu, you will see an error page, as shown in Figure 9. The error message is shown because there are no controllers in the application to handle requests at the moment, which is something that we will address shortly.

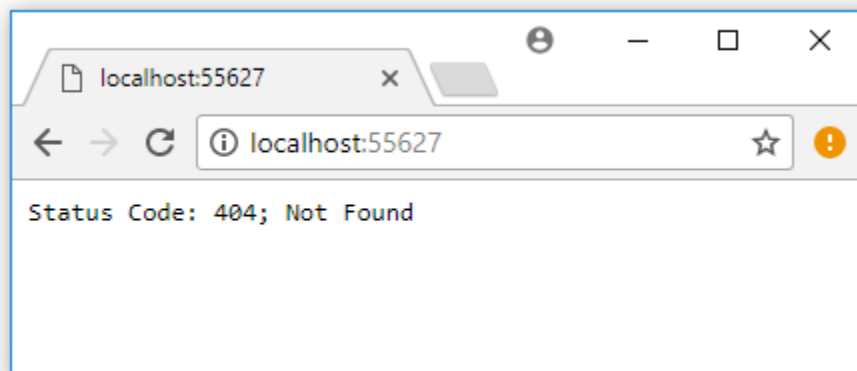


Figure 10. Running the SportsStore application

1.7 Task 7: Creating Product 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 we need is a product. We need to create a class file called `Product.cs` to the Models folder and define the class as shown in Listing 4.

To create the class file, right-click the Models folder in the Solution Explorer and select **Add ► Class** from the pop-up menu. Set the name of the new class to `Product.cs` and click the Add button. Edit the contents of the new class file to match Listing 4.

Listing 4. The Contents of the `Product.cs` File in the Models Folder

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

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; }
    }
}
```

Note:

Now that we created a model along with `SportsStore.Models` namespace, the error in the `_ViewImports` file would be resolved.

1.8 Task 8: Creating a Product Repository Interface

We need to define an interface class to provide meaningful ways of storing and retrieving domain model data. To create the interface file, right-click the Models folder in the Solution Explorer and select Add ► Class from the pop-up menu. In the popup window, select Interface and set the name of the new file to IProductRepository.cs and click the Add button, as shown in Figure 11. Edit the contents of the new class file to match Listing 5.

Listing 5. The Contents of the IProductRepository.cs File in the Models Folder

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

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. We will revisit the `IProductRepository` interface throughout the development process to add features.

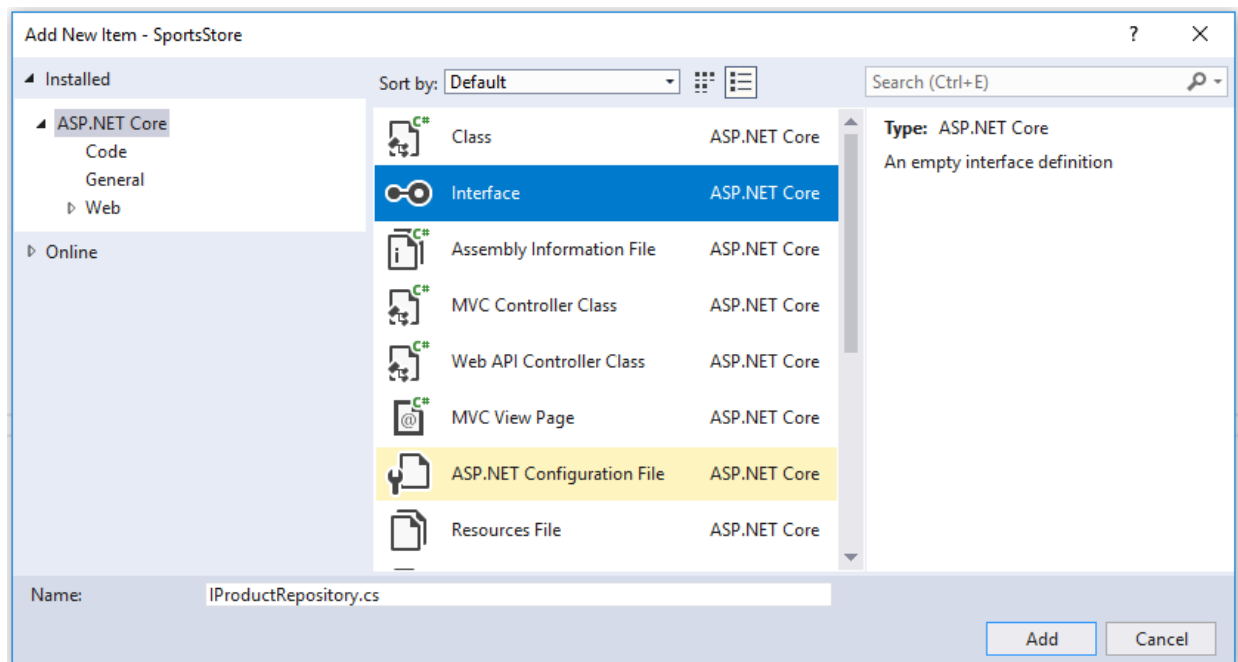


Figure 11. Adding an Interface file

Note:

We create a repository interface to act as a mediatory between data source layers that handles data logic and domain models that handles business logic. Separation of logic provide us ability to add more logic and grow the application feature sets. Repository interface will be used for mapping the data from a business entity to the data source, querying the data source for the data, and persists changes in the business entity to the data source. You can learn more about the repository pattern at <https://msdn.microsoft.com/en-us/library/ff649690.aspx>.

1.9 Task 9: Creating a Fake Repository

The model includes the logic for storing and retrieving the data from the persistent data store. We need some way of getting Product objects from a database. As we have not discussed database, we will rather create a fake implementation of the IProductRepository interface that will stand in until we return to the topic of data storage. To create the fake repository, add a class file called FakeProductRepository.cs to the Models folder and define the class as shown in Listing 6. The FakeProductRepository class implements the IProductRepository interface by returning a fixed collection of Product objects as the value of the Products property.

Listing 6. The Contents of FakeProductRepository.cs File in the Models Folder

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

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 }
        };
    }
}
```

1.10 Task 10: 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.

For the SportsStore application, we need 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 us to start developing the application using the simple FakeProductRepository class 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 7, we define a new service for the repository. To edit the Startup class, double click on the Startup.cs file available in the solution explorer.

Listing 7. Creating the Repository Service in the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;

namespace SportsStore
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add services to the
        // container.
        // For more information on how to configure your application, visit
        // https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddTransient<IProductRepository, FakeProductRepository>();

            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        // pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvcWithDefaultRoute();
        }
    }
}
```

The statement 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.

1.11 Task 11: Creating Controller for Displaying a List of Products

In this section, we are 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. We will also set up an initial routing configuration so that MVC knows how to map requests for the application to the controller.

To create the first controller in the application, add a class file called `ProductController.cs` to the `Controllers` folder and define the class as shown in Listing 8.

Listing 8. The Contents of the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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, we will replace the fake repository with the real one, and dependency injection means that the controller will continue to work without changes.

Next, we will add an action method, called List, which will render a view showing the complete list of the products in the repository, as shown in Listing 9. 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.

Listing 9. Adding an Action Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers
{
    public class ProductController : Controller
    {
        private IProductRepository repository;
        public ProductController(IProductRepository repo)
        {
            repository = repo;
        }
    }
}
```

```

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

```

1.12 Task 12: Adding and Configuring the Layout of the View

In an ASP.NET Core MVC application, a component called the view engine is used to produce the content sent to clients. The default view engine is called Razor, and it process annotated HTML files for instructions that insert dynamic content into the output sent to the browser.

We 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.

Layouts are typically shared by views used by multiple controllers and are stored in a folder called Views/Shared, which is one of the locations that Razor looks in when it tries to find a file. To create Shared folder under Views, select and right click on Views folder, and Add ► New Folder from the pop-up menu. Rename the folder as Shared.

To create a layout, select and right click on Views/Shared folder, and select Add ► New Item from the pop-up menu. Select the MVC View Layout Page template from the ASP.NET category and set the file name to _Layout.cshtml, as shown in Figure 12, which is the default name that Visual Studio assigns to this item type. Click the Add button to create the file. (Names of layout files begin with an underscore.) Listing 10 shows the _Layout.cshtml file. We make one change to the default content, which is to set the contents of the title element to SportsStore.

Listing 10. 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>

```

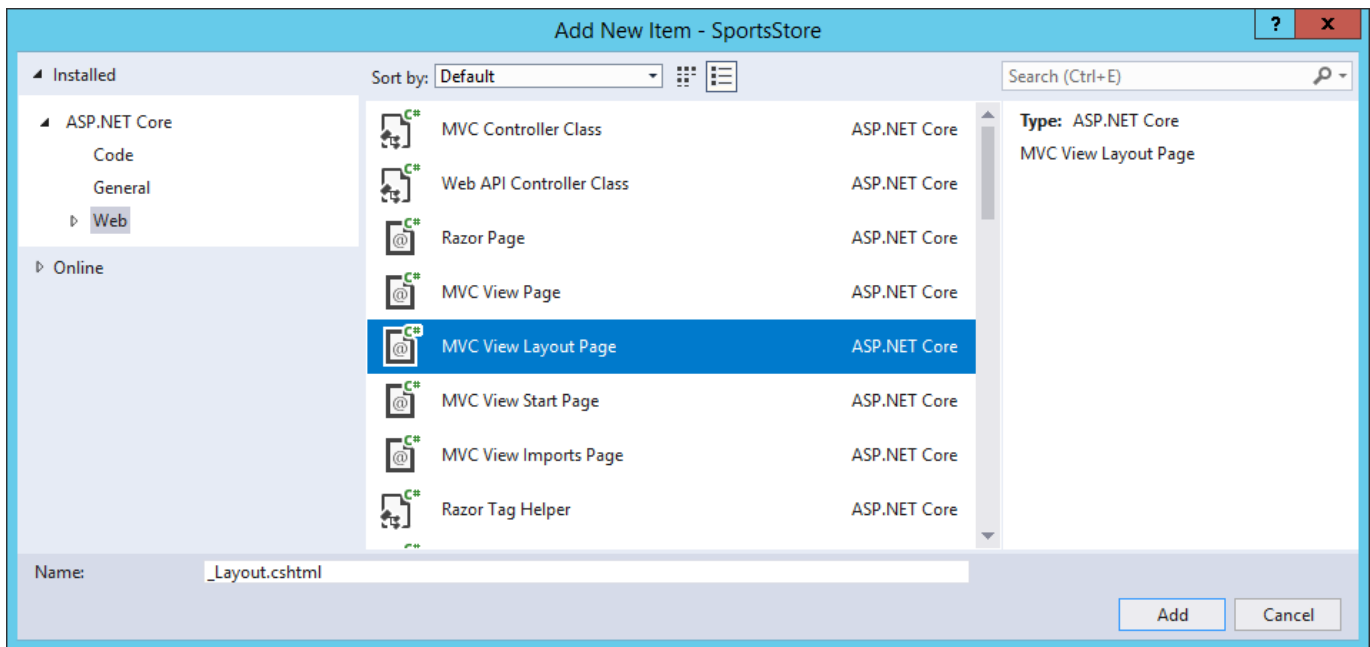


Figure 12. Creating a layout

1.13 Task 13: Adding and Configuring ViewStart

Next, we 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, as shown in Figure 13. The default content added by Visual Studio, shown in Listing 11, selects a layout called `_Layout.cshtml`, which corresponds to the file shown in Listing 11.

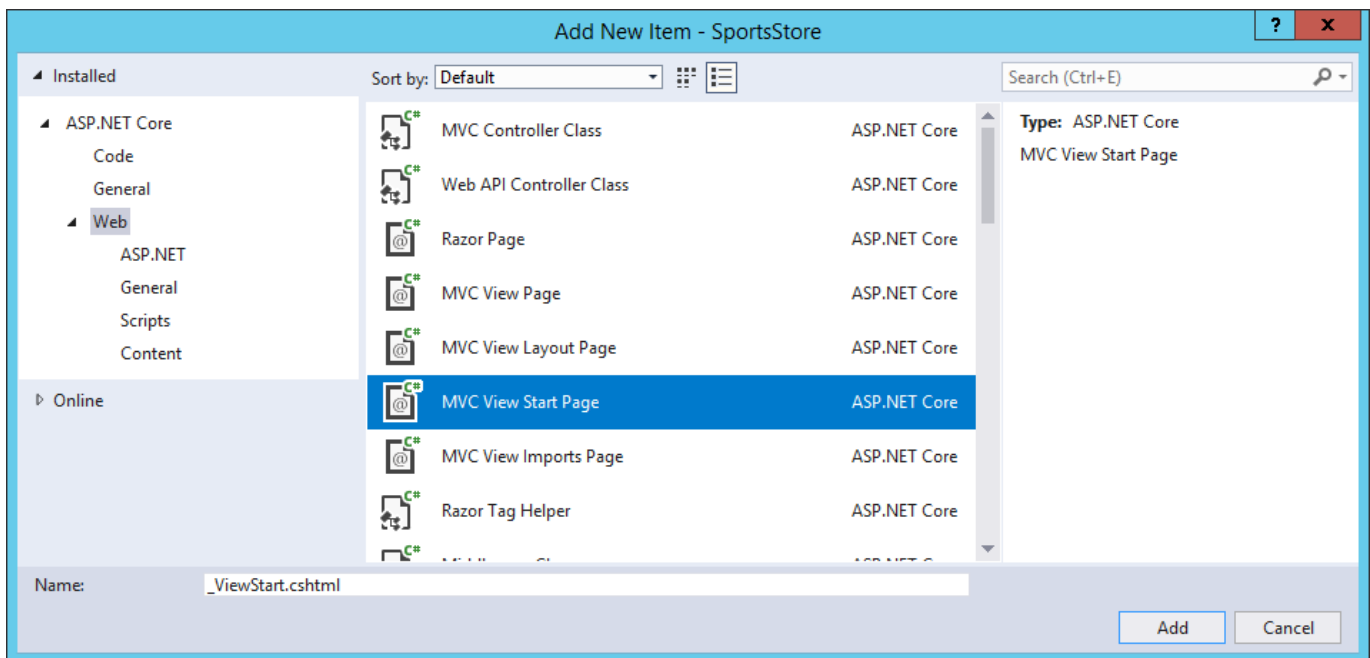


Figure 13. Creating View Start Page

Listing 11. The Contents of the _ViewStart.cshtml File in the Views Folder

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

1.14 Task 14: Adding View to Display Product Listing

Now we need to add the view that will be displayed when the List action method is used to handle a request. First, we need to create the Views/Product folder. For creating Product folder, select and right click on Views folder, and Add ► New Folder from the pop-up menu. Rename the folder as Product. After that, select and right click on Product folder, select Add ► New Item from the pop-up menu. Select the MVC View Page and set the file name to List.cshtml, as shown in Figure 44. We then add the markup shown in Listing 12.

Listing 12. 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  
        <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. We 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 principle.

Note:

Tip we 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.

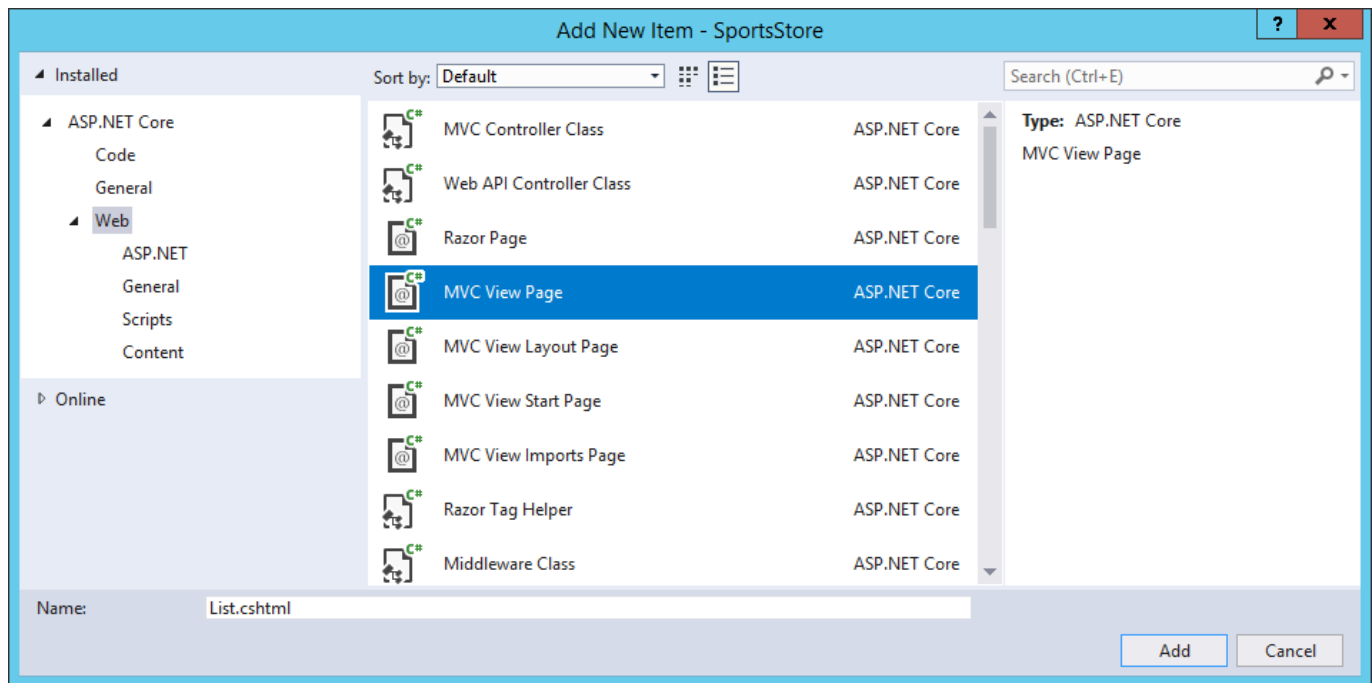


Figure I4. Creating View Page

1.15 Task 15: Setting the Default Route

We need to tell MVC that it should send requests that arrive for the root URL of the application (<http://mysite/>) to the List action method in the ProductController class. We do this by editing the statement in the Startup class that sets up the MVC classes that handle HTTP requests, as shown in Listing I3.

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. The change in Listing I3 tells MVC to send requests to the List action method of the Product controller unless the request URL specifies otherwise.

Note: We have set the name of the controller in Listing I3 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.

Listing I3. Changing the Default Route in the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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
    {
        // This method gets called by the runtime. Use this method to add services to the
        container.
        // For more information on how to configure your application, visit
        https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddTransient<IProductRepository,
                FakeProductRepository>();
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        pipeline.
        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?}");
            });
        }
    }
}

```

1.16 Task 16: Running the Application

All the basics are in place. We 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 15.

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.

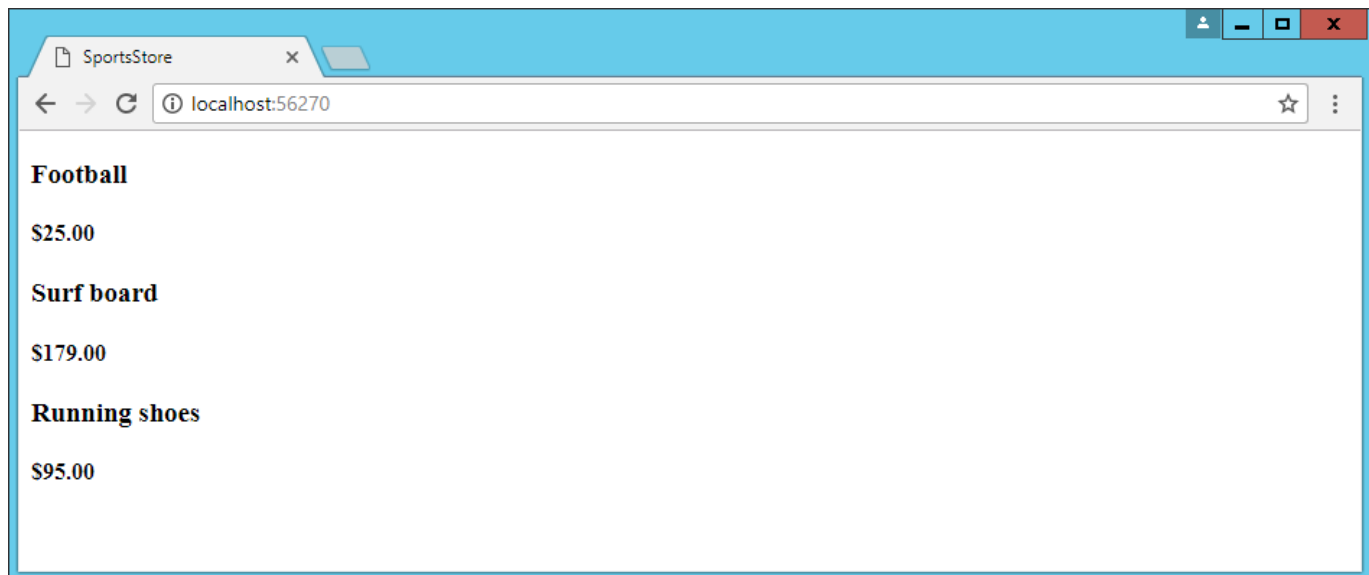


Figure 15. Viewing the basic application functionality

1.17 Preparing a Database

We can display a simple view that contains details of the products, but it uses the test data that the fake repository contains. Before we can implement a real repository with real data, we need to set up a database and populate it with some data.

We are going to use SQL Server as the database, and we 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. We are using Entity Framework Core as it is simple to get working, and it works nicely with ASP.NET Core MVC. You learn more about the Entity Framework Core at <https://docs.microsoft.com/en-us/ef/core/>.

1.18 Task 17: Installing Entity Framework Core

Entity Framework Core is installed using NuGet and Listing 14 shows the additions that are required to the SportsStore.csproj file in the SportsStore application 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 14. When you save the SportsStore.csproj file, Visual Studio will download and install EF Core and add it to the project.

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.

Note: 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 14. Adding Entity Framework in the SportsStore.csproj File

```
</Project>

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

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

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />
    <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="2.0.1" />

    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="2.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="2.0.1" />
    <PrivateAssets>"All" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.0.1" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
      Version="2.0.1" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.1" />
  </ItemGroup>
</Project>
```

1.19 Task 18: 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, add a class file called `ApplicationDbContext.cs` to the Models folder and define the class as shown in Listing 15.

Listing 15. The Contents of the ApplicationDbContext.cs File in the Models Folder

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

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.

1.20 Task 19: 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. We add a class file called EFProductRepository.cs to the Models folder and use it to define the repository class shown in Listing 16.

We will add additional functionality as we 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.

Listing 16. The Contents of the EFProductRepository.cs File in the Models Folder

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace SportsStore.Models
{
    public class EFProductRepository : IProductRepository
    {
        private ApplicationDbContext context;
        public EFProductRepository(ApplicationDbContext ctx)
        {
            context = ctx;
        }
        public IEnumerable<Product> Products => context.Products;
    }
}

```

1.21 Task 20: Obtaining the Database Connection String

Each student in this class has access to their own database created for practicing this tutorial. A Microsoft SQL Server database has been created at the Server instance: CISVM-DBMS-1.unfcsd.unf.edu\COP3855. Your database will be named using your N-number. In order to connect, the web application with the database, we need connection string information for your database. Follow below steps to obtain the connection string:

1. Click on the View menu from the Main menu options in the Visual Studio, then select SQL Server Object Explorer option, as shown in the Figure 16.
2. In the SQL Server Object Explorer, click on the Add SQL Server option. See highlighted section in the Figure 17.
3. In the Connect window, expand the Network list, select "CISVM-DBMS-1\COP3855" option. Ensure Windows Authentication is selected and "UNFCSD\your n-number" is displayed, as shown in Figure 18.
4. Click Connect button in the Connect window.

5. In the SQL Server Object Explorer, expand CISVM-DBMS and find your N-number database. See Figure 19 for expanded view.
6. When you select your database, in the Property window in the Visual Studio, properties of your database will be displayed. Property window are typically right-hand bottom corner of the Visual Studio.
7. In the Property window, there will be value for Connection string, as shown in the Figure 20. You can copy the connection string into a notepad. We will need this value for our next task. Your connection string will look quite similar to below but with your N-number on it.

Data Source=CISVM-DBMS-1\COP3855;Initial Catalog=n00611704;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailover=False

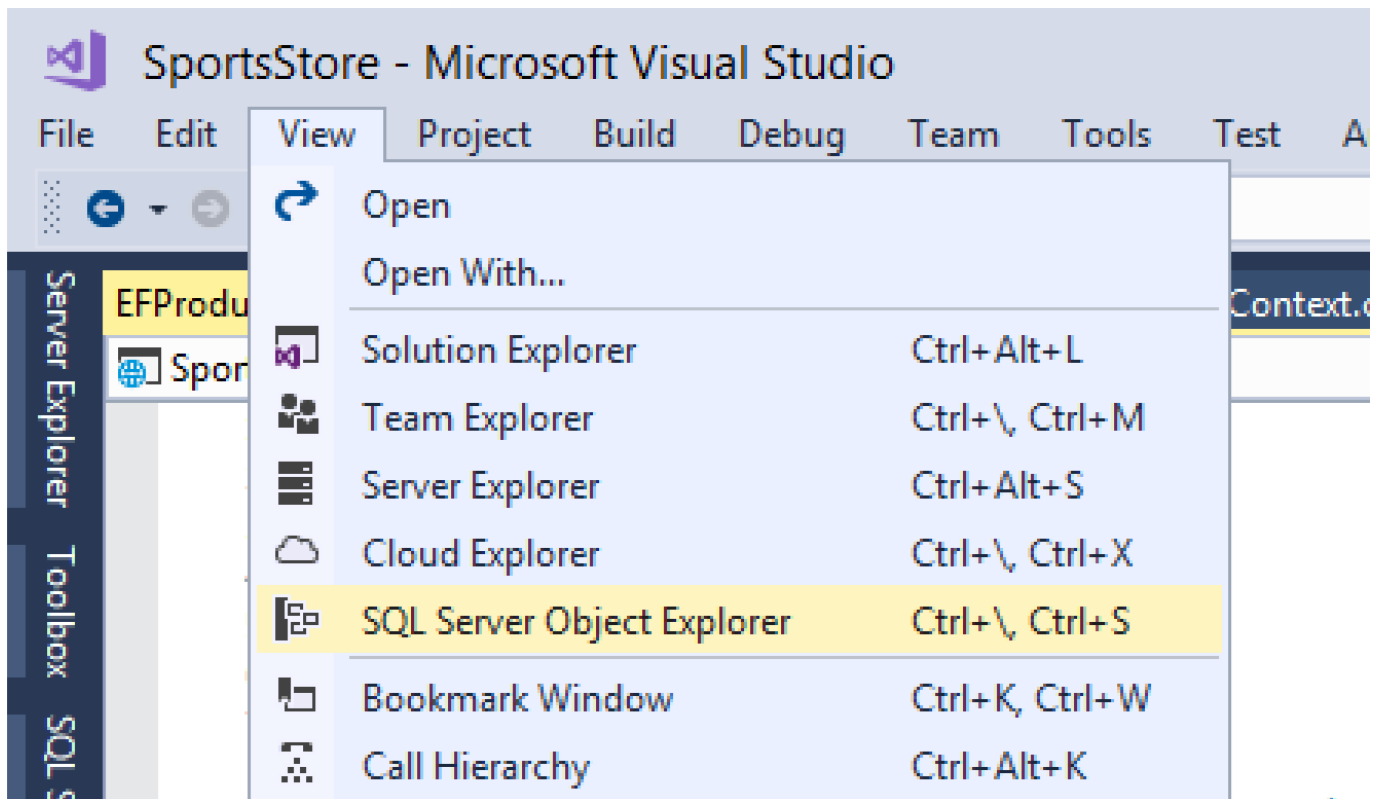


Figure 16. Opening SQL Server Object Explorer

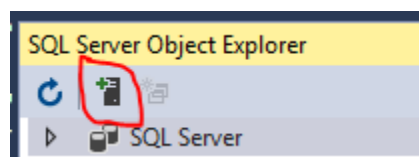


Figure 17. Adding SQL Server to the project

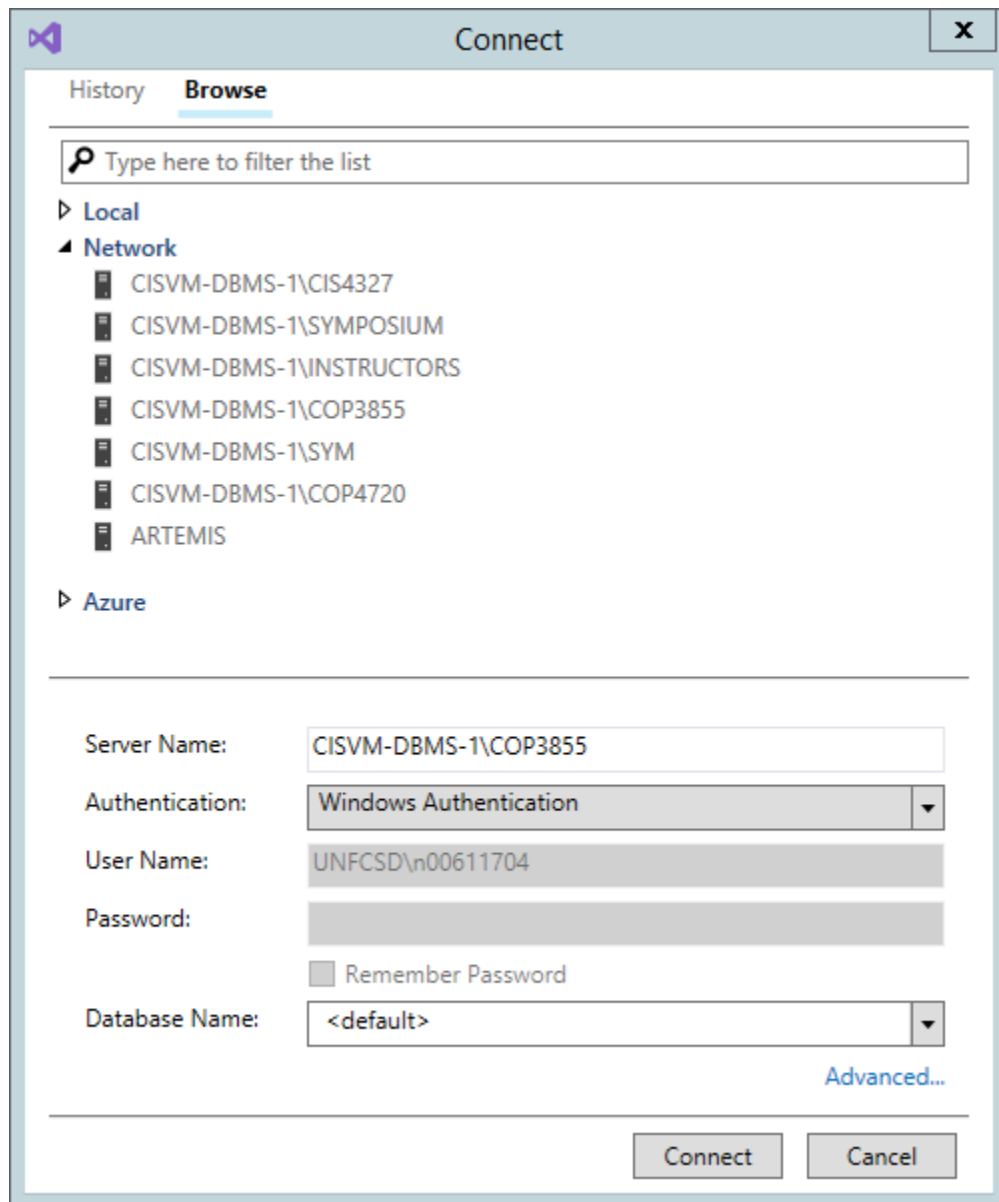


Figure I8. SQL Server connection configurations

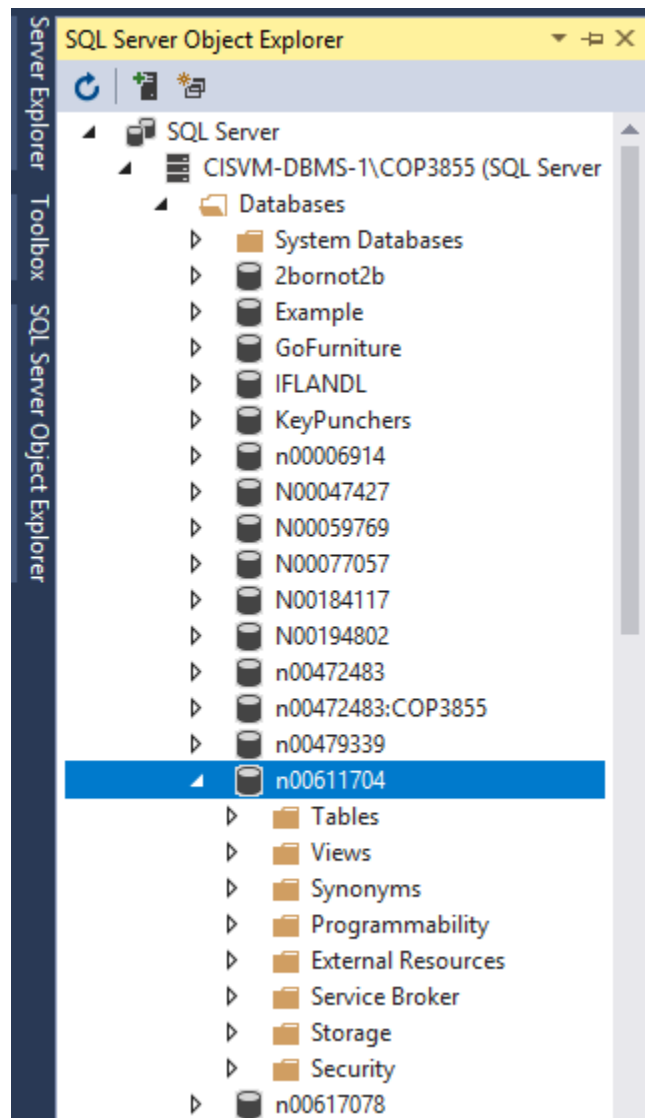


Figure 19. View your N-number database in the SQL Server Object Explorer

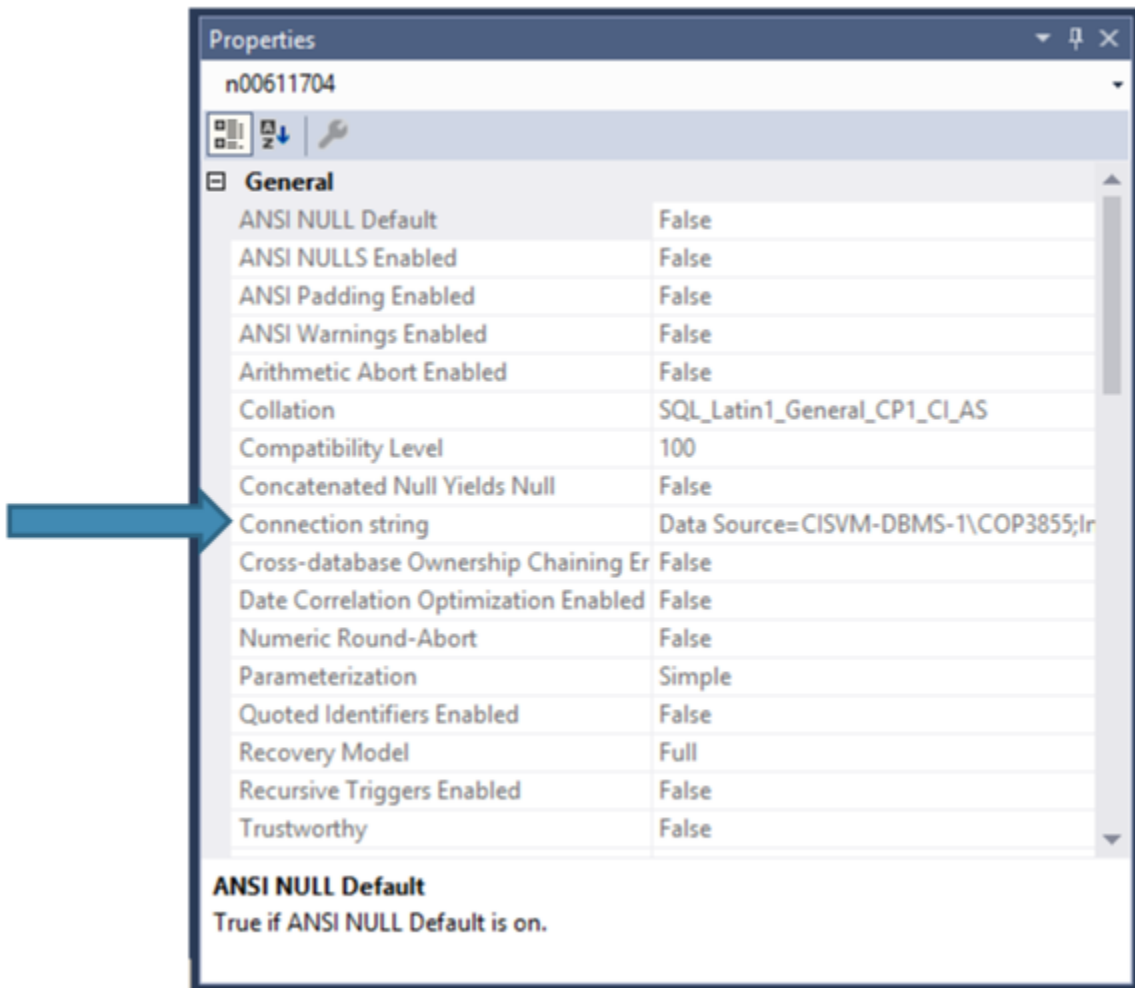


Figure 20. Viewing database properties

1.22 Task 21: 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 we need to create for the SportsStore project using the ASP.NET Configuration File item template in the ASP.NET section of the Add New Item window.

To add `appsettings.json` file to the project, select SportsStore in the Solution window, then select Add ➤ New Item from the pop-up menu, as shown in the Figure 21. Select the ASP.NET Configuration File and set the file name to `appsettings.json`, as shown in Figure 22.

Visual Studio adds a placeholder connection string to the `appsettings.json` file when it creates the file, which we edit it and paste the connection string of your database as shown in the Listing 17.

Listing 17. Editing the Connection String in the appsettings.json File

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=CISVM-DBMS-1\\COP3855;Initial Catalog=n00611704;Integrated
Security=True;Connect
Timeout=30;Encrypt=False;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailo
ver=False"
  }
}
```

Note: 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 17. When you define the connection string in your own project, make sure that the value of the ConnectionString item is on a single line.

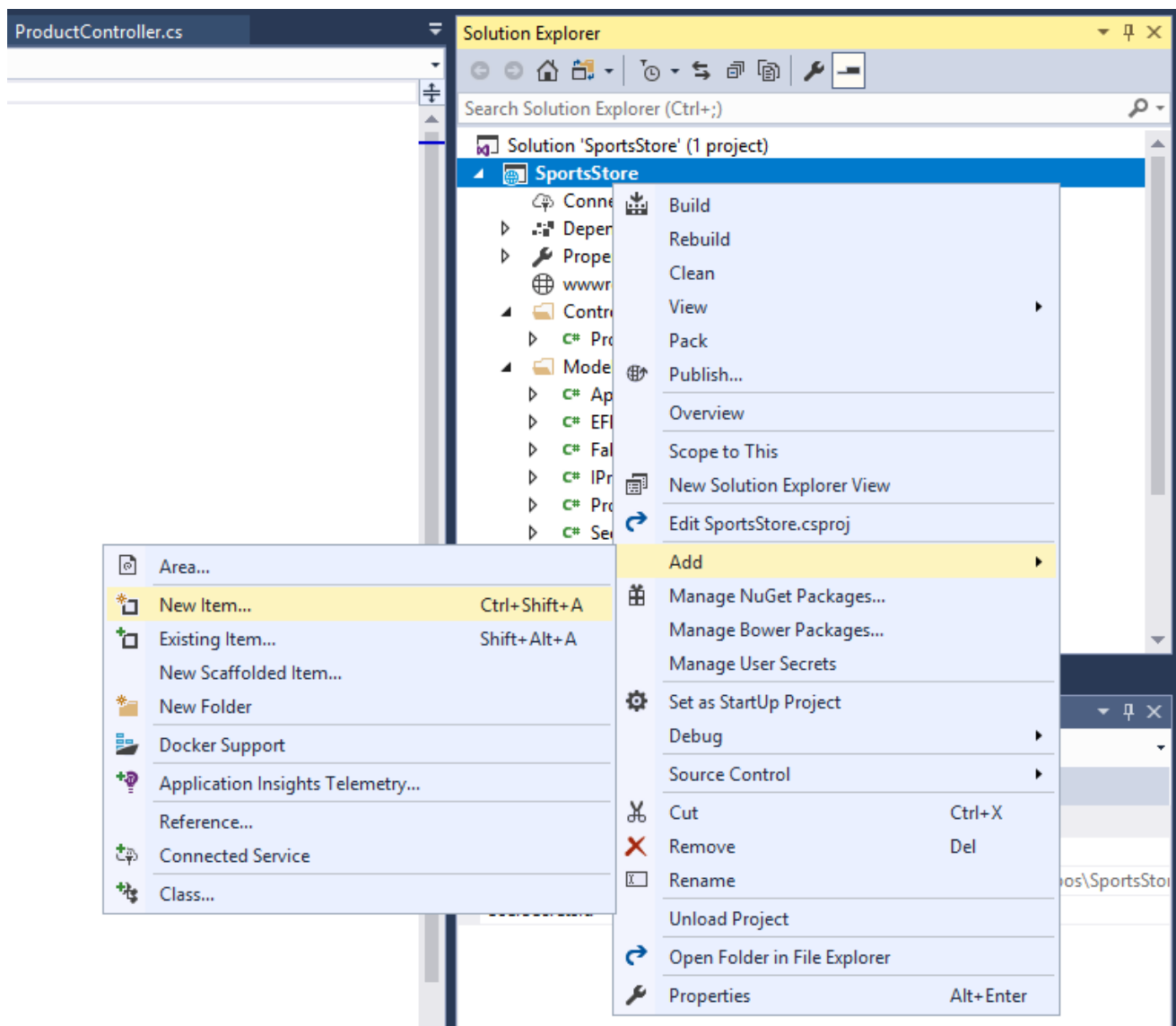


Figure 21. Adding new item to the project solution

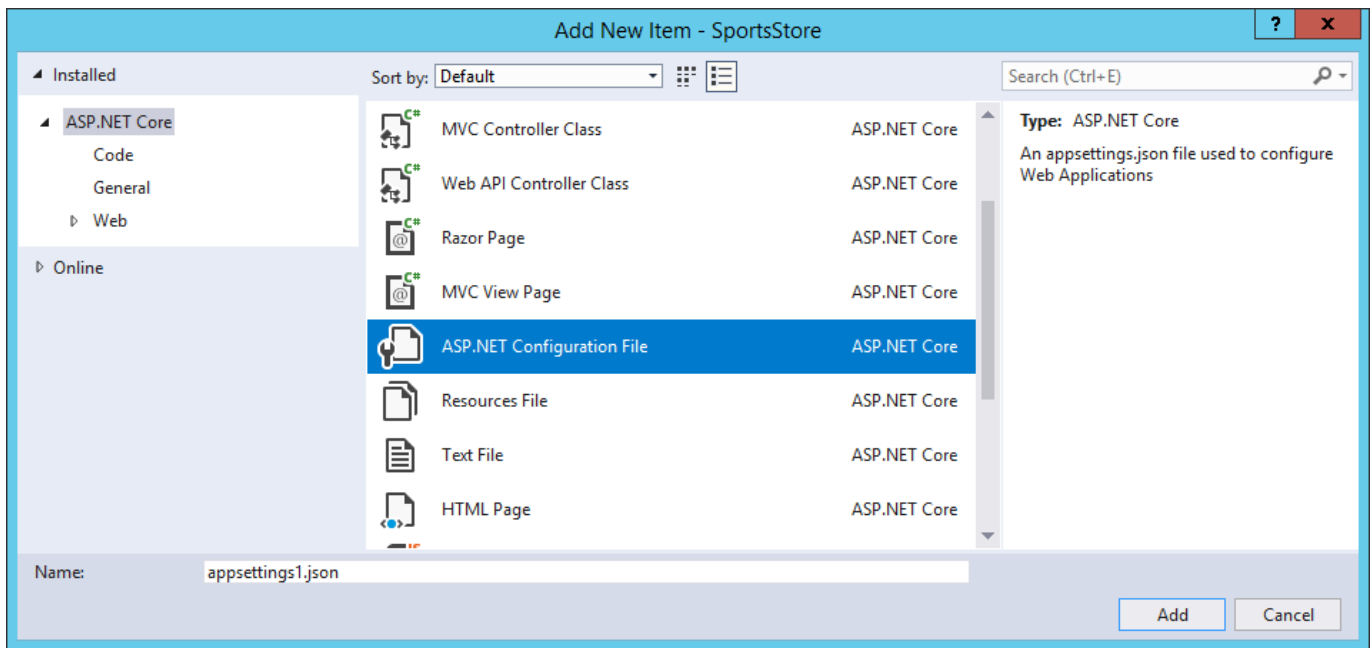


Figure 22. Adding appsettings.json to the project solution

1.23 Task 22: 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 18 shows the change to the SportsStore.csproj file.

Listing 18. Adding a Package in the SportsStore.csproj File

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

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

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />
    <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="2.0.1" />

    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="2.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="2.0.1" />
    PrivateAssets="All" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.0.1" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="2.0.1" />
  </ItemGroup>

  <ItemGroup>
```

```

        <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
Version="2.0.1" />
        <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version=" 2.0.1"
/>
    </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 19.

Listing 19. Configuring the Application in the Startup.cs File

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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();
        }

        // This method gets called by the runtime. Use this method to add services to the
        // container.
        // For more information on how to configure your application, visit
        // https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        // pipeline.
        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 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. Within the ConfigureServices method, we have added a sequence of method calls that sets up Entity Framework Core.

The AddDbContext extension method sets up the services provided by Entity Framework Core for the database context class. We configure the database with the UseSqlServer method and specified the connection string, which is obtained from the Configuration property.

The next change we make in the Startup class was to replace the fake repository with the real one. 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. The fake data will be seamlessly replaced by the real data in the database without having to change the ProductController class.

1.24 Task 23: 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. You can learn more about migrations at <https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/>.

EF Core commands are performed using the Package Manager Console, which you can open through the Visual Studio Tools > NuGet Package Manager menu as shown in Figure 23.

Run the following command in the Package Manager Console to create the migration class that will prepare the database for its first use (as shown in Figure 24):

Add-Migration Initial

When this command has finished, you will see a Migrations folder in the Visual Studio Solution Explorer window as shown in Figure 25. 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.

Note: 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.

While running update database command, in case, you get below error message you can ignore for the moment as shown in the Figure 26.

```
An error occurred while calling method 'BuildWebHost' on class 'Program'. Continuing without the application service provider. Error: Cannot resolve scoped service 'SportsStore.Models.ApplicationDbContext' from root provider.
```

If the database was updated successfully, you should be able to see Products table in your N-number database in the SQL Server Object Explorer as shown in the Figure 27. You may have to refresh the object explorer to view updates to the database. When you double click the Product table, you can view table design details as shown in the Figure 28.

Product table is now empty. We need to add records to it. For now, we are going to manually enter Product table values. We will create view for entering values into the table later. Select the Product table, right click, and the select View Data option, as shown in the Figure 29.

Visual Studio will show you the Product table record, which will be empty. You can manually enter values into the table as shown in the Figure 30. Enter values for columns: Category, Description, Name, and Price. Database adds Product ID information as you enter record values. Based on Figure 30, go ahead and add product details to your table.

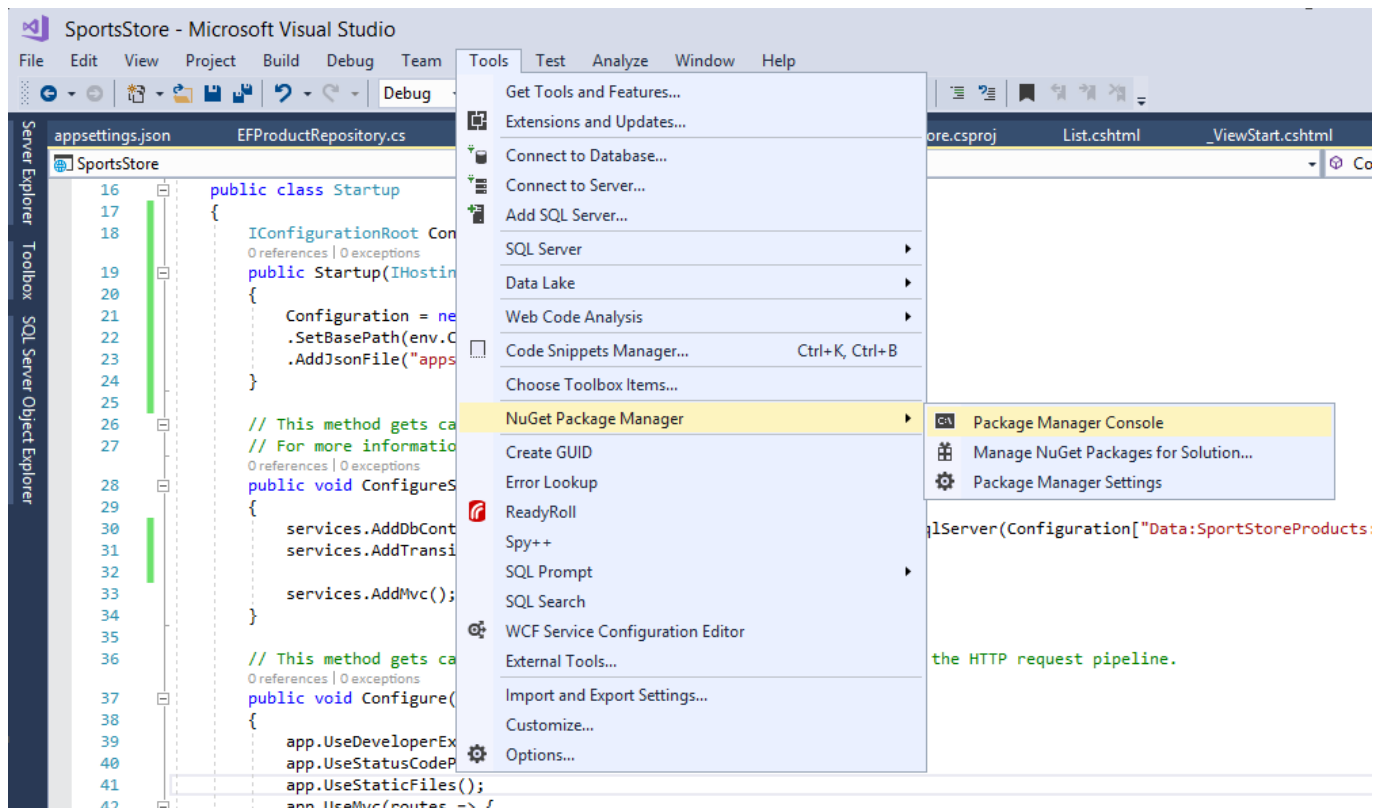


Figure 23. Selecting NuGet Package Manager Console

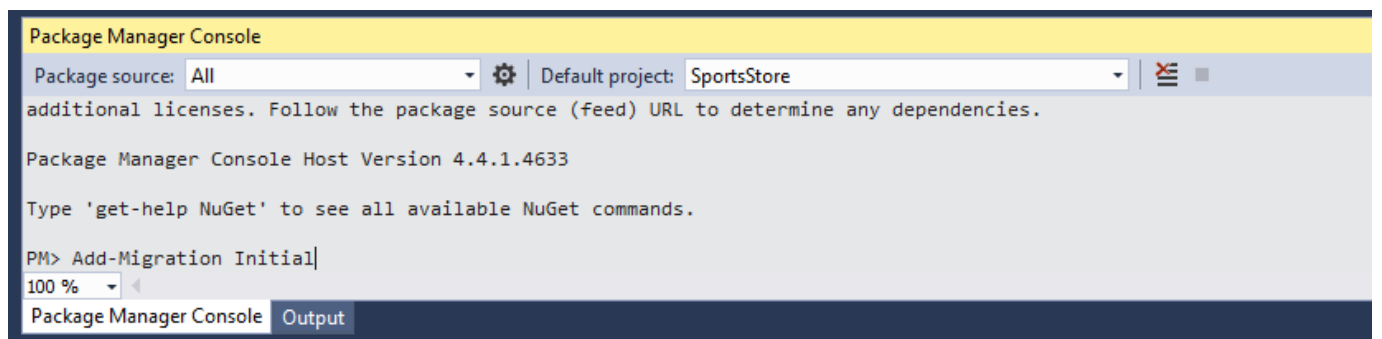


Figure 24. Running commands in NuGet Package Manager Console

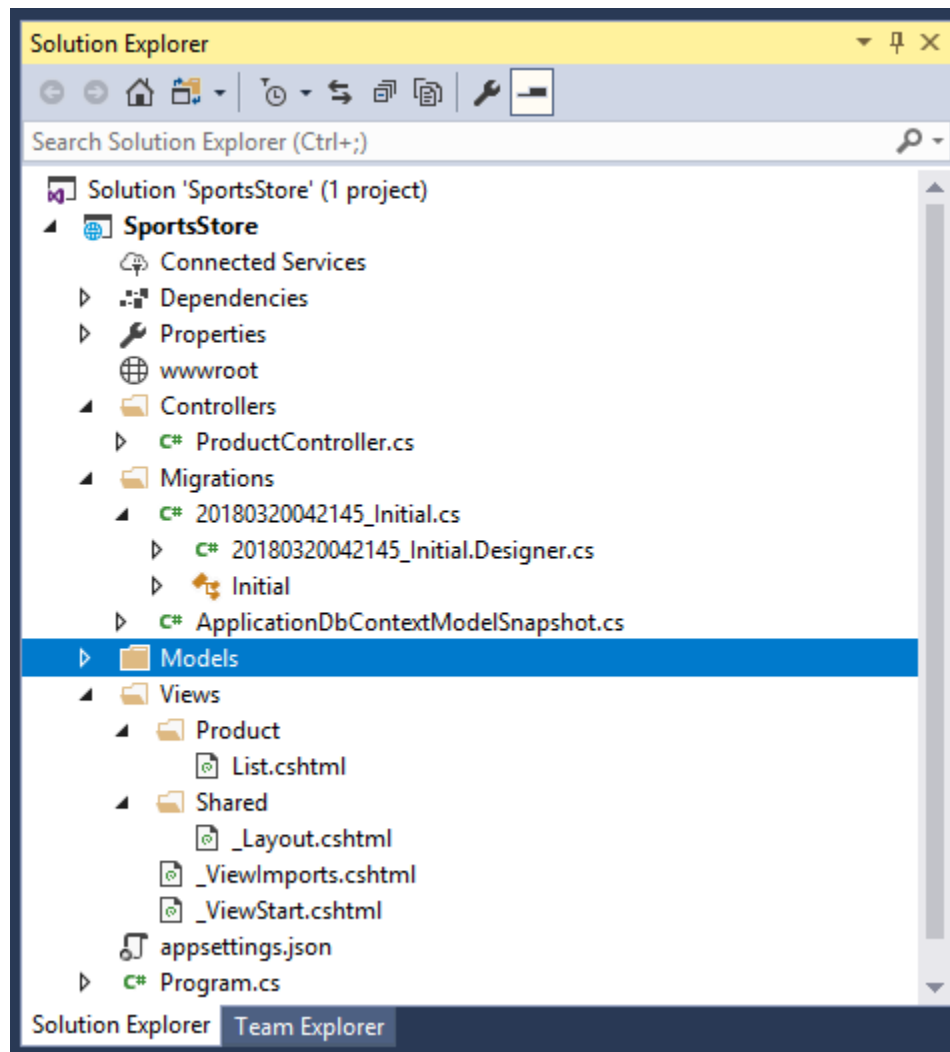


Figure 25. Solution Explorer view with migrations folder

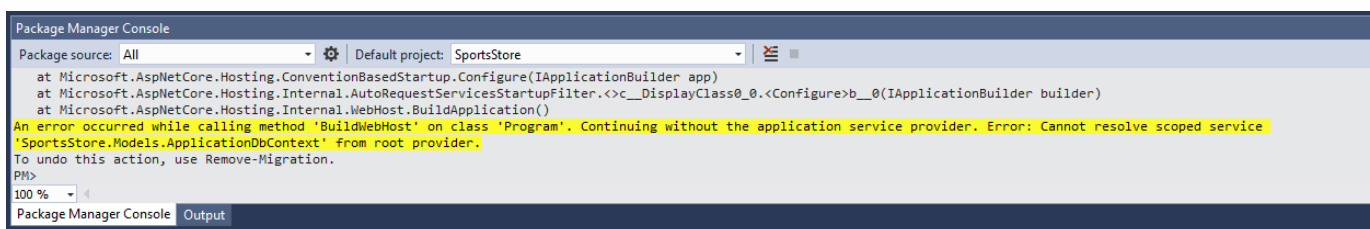


Figure 26. ApplicationDbContext Error in Program.cs file

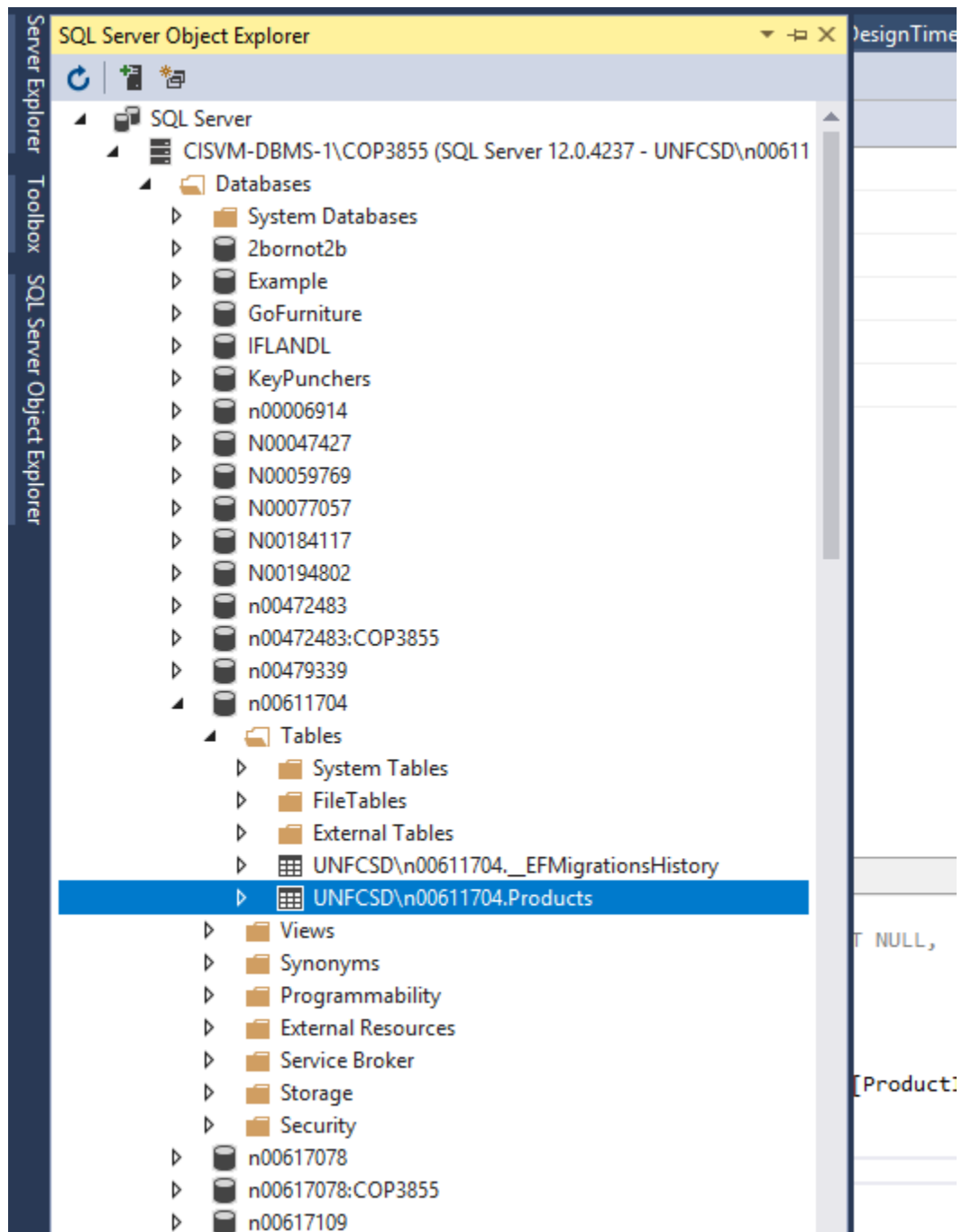


Figure 27. View Products Table in the Server Server Object Explorer

UNFCSD\n00611704.Products [Design] 20180320042145_Initial.cs					
Update Script File: UNFCSD_n00611704.Products.sql					
	Name	Data Type	Allow Nulls	Default	
	ProductID	int	<input type="checkbox"/>		
	Category	nvarchar(MAX)	<input checked="" type="checkbox"/>		
	Description	nvarchar(MAX)	<input checked="" type="checkbox"/>		
	Name	nvarchar(MAX)	<input checked="" type="checkbox"/>		
	Price	decimal(18,2)	<input type="checkbox"/>		
			<input type="checkbox"/>		

Figure 28. View Products Table Design

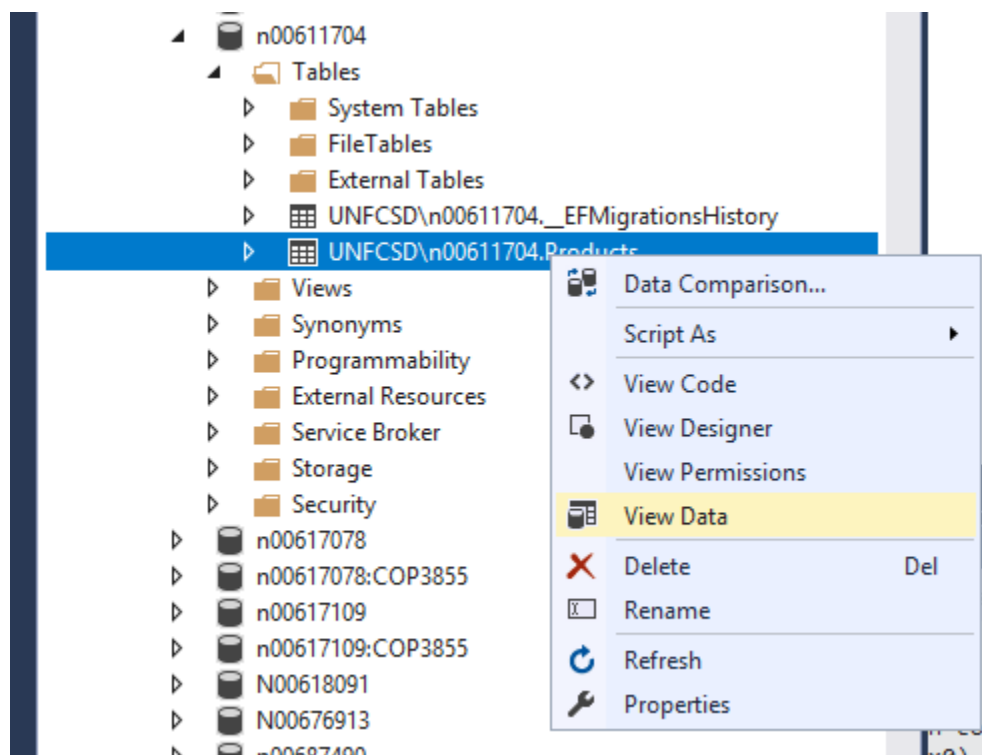


Figure 29. Use View Data to edit Table records

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

Figure 30. Entering records into the Product Table

1.25 Task 24: Running the Application

Run the application by clicking on the Run IIS Express option. 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 31.

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 us to focus on ASP.NET Core MVC. We did skip over a lot of the detail in how EF Core operates and the huge number of configuration options that are available. 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>.

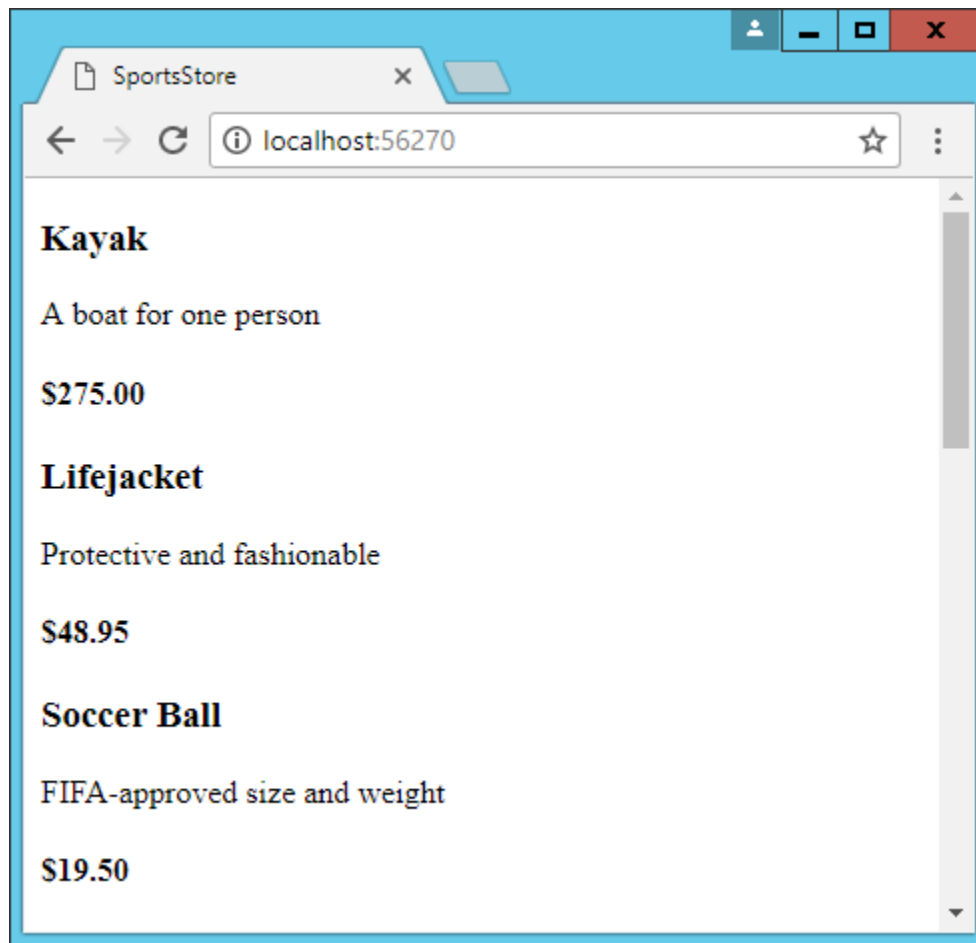


Figure 31. SportsStore display using the database repository

1.26 Task 25: Adding Pagination

In the figure 31, you can see that List.cshtml file displays the products in the database on a single page. Pages with listing of items should be displayed using pagination to make navigation of items easier for users. In this section, we 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, we are going to add a parameter to the List method in the Product controller, as shown in Listing 20.

Listing 20. Adding Pagination Support to the List Action Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers
{
    public class ProductController : Controller
    {
```

```

private IProductRepository repository;
public int PageSize = 4;

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

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

```

The `PageSize` field specifies that we want four products per page. We will replace this with a better mechanism later. We added an optional parameter to the `List` method. This means that if we call the method without a parameter (`List()`), call is treated as though we 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, we 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.

If you run the application, you will see that there are now four items shown on the page, see Figure 32. If you want to view another page (see Figure 33 for second page), you can append query string parameters to the end of the URL, like this: “/?page=2”

The full URL would like this: **`http://localhost:56270/?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, we need to render some page links at the bottom of each list of products so that customers can navigate between pages. To do this, we are going to implement a tag helper, which generates the HTML markup for the links.

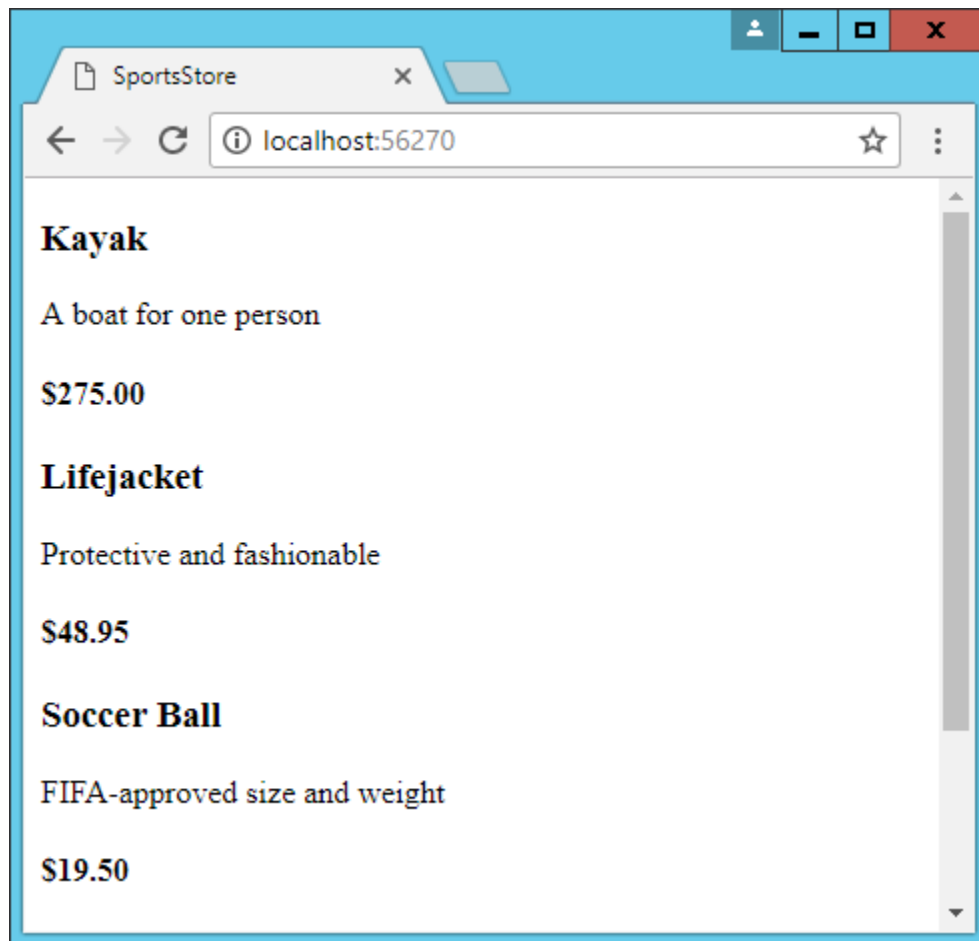


Figure 32. First page of SportsStore Pagination display

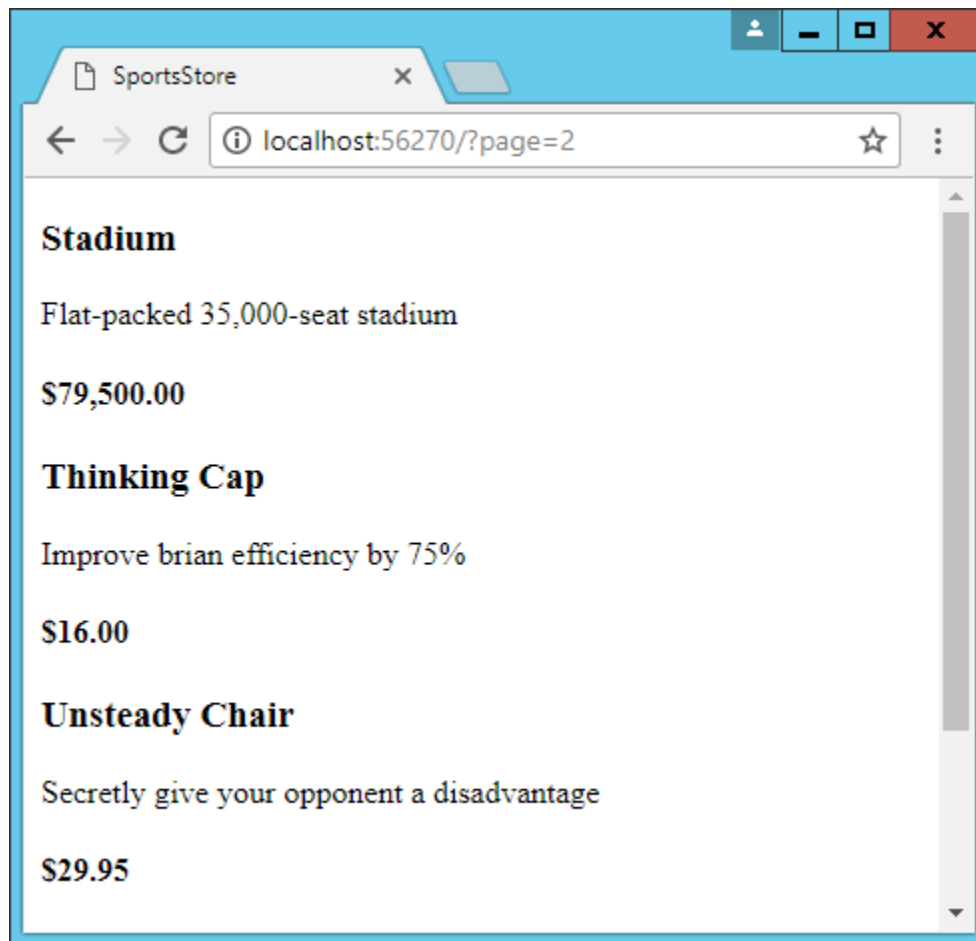


Figure 33. Second page of SportsStore Pagination display

1.27 Task 26: Adding the View Model

To support the tag helper, we are 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. We create a `Models/ViewModels` folder in the `SportsStore` project as shown in Figure 34. Add a class file called `PagingInfo.cs` to the `ViewModels` folder as defined in Listing 21.

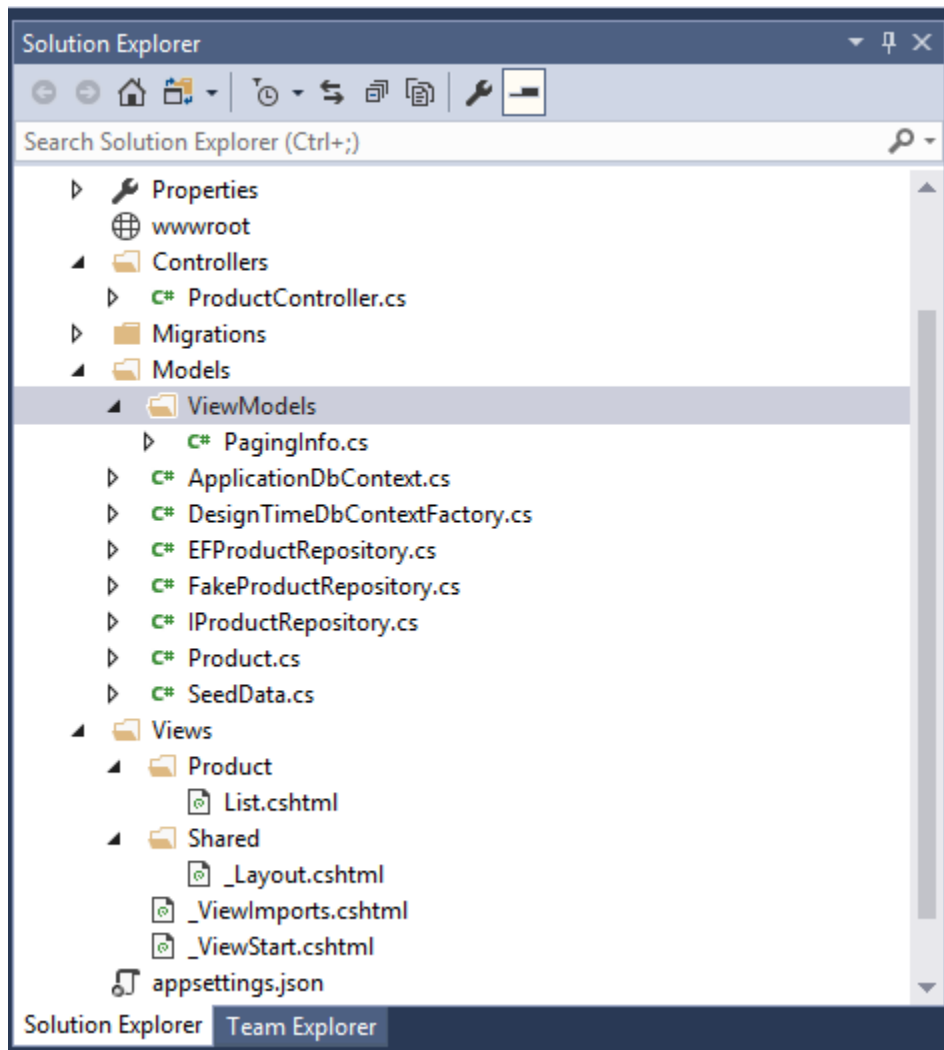


Figure 34. Solution explorer with Models/ViewModels folder in the SportsStore project

Note: Differences between Models and ViewModels:

Model represents the domain objects and its behaviors in relevance to the application context. Classes created as Models helps us to manage data, logic, and rules associated with domain objects independent of the user interface. We use these domain object model classes to send information to the database (save, create, update, and delete records), to perform business calculations, and to render a view. Usually we put all our model classes in the Model folder.

ViewModel is very similar to a "model". The major difference between "Model" and "ViewModel" is that we use a ViewModel when view rendered need to differ from domain Model. In these cases, in addition to Model class, ViewModel class is created where what needs to be rendered in the view is described. We put all our ViewModel classes in a "ViewModels" named folder.

Here is an example scenario:

Say that, we want a user to create their login information for their account, using username and password. The Login class in the Model folder would be following:

```
public class Login
{
    public String Username { get; set; }
    public String Password { get; set; }
}
```

Let's assume we want to implement a view page that will have three textboxes for Username, Password and Re-enter Password. In this case, we want user to re-enter the password for confirmation purposes. Generation of second password entry in the view makes it different from what is described in the Login class. In this case, we will need to create a LoginViewModel class in the Models/ViewModels folder to describe view that need to be generated. The Login class in the LoginViewModel folder would be following:

```
public class LoginViewModel
{
    public String Username { get; set; }
    public String Password { get; set; }
    public String RePassword { get; set; }
}
```

If we don't take the advantage of a ViewModel, then we would be unnecessarily recording the RePassword in the database.

You can learn more about the Model-View-ViewModel pattern at <https://msdn.microsoft.com/en-us/library/448246.aspx>.

Listing 21. The Contents of the PagingInfo.cs File in the Models/ViewModels Folder

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

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);
    }
}
```

1.28 Task 27: Adding the Tag Helper Class

Now that we have a view model, we can create a tag helper class. Before we create a tag helper class, let's understand what tag helpers are.

Tag helpers are a new feature that has been introduced in ASP.NET Core MVC and are C# classes that transform HTML elements in a view. Common uses for tag helpers include generating URLs for forms using the application's routing configuration, ensuring that elements of a specific type are styled consistently, and replacing custom shorthand elements with commonly used fragments of content. Table 3 puts tag helpers in context. You can learn more about the Tag Helpers at <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro>.

Table 3. Putting Tag Helpers in Context

Question	Answer
What are they?	Tag helpers are classes that manipulate HTML elements, either to change them in some way, to supplement them with additional content, or to replace them entirely with new content.
Why are they useful?	Tag helpers allow view content to be generated or transformed using C# logic, ensuring that the HTML sent to the client reflects the state of the application.
How are they used?	The HTML elements to which tag helpers are applied are selected based on the name of the class or through the use of the <code>HTMLTargetElement</code> attribute. When a view is rendered, elements are transformed by tag helpers and included in the HTML sent to the client.
Are there any pitfalls or limitations?	It can be easy to get carried away and generate complex sections of HTML content using tag helpers, which is something that is more readily achieved using view components.
Are there any alternatives?	You don't have to use tag helpers, but they make it easy to generate complex HTML in MVC applications.
Have they changed since MVC 5?	Tag helpers are a new addition in ASP.NET Core MVC and replace the functionality provided by HTML helpers.

We create the Infrastructure folder in the SportsStore project (see Figure 35) and add to it a class file called `PageLinkTagHelper.cs`, which we use to define the class shown in Listing 22. This tag helper populates a `div` element with `a` elements that correspond to pages of products.

Note: The Infrastructure folder is where we put classes that deliver the plumbing for an application but that are not related to the application's domain.

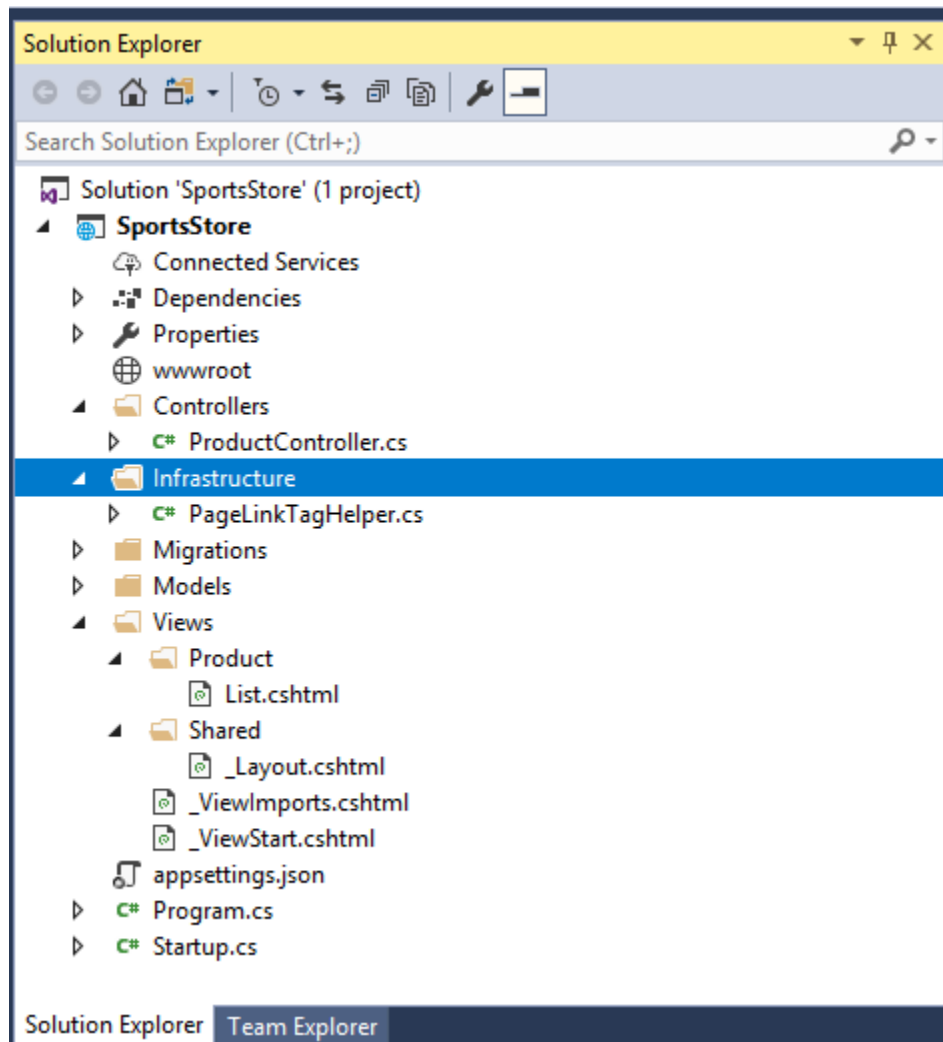


Figure 35. Solution explorer with Infrastructure folder in the SportsStore project

Listing 22. 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)
{
    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 });
        tag.InnerHtml.Append(i.ToString());
        result.InnerHtml.AppendHtml(tag);
    }
    output.Content.AppendHtml(result.InnerHtml);
}
}
}

```

Most MVC components, such as controllers and views, are discovered automatically, but tag helpers have to be registered. In Listing 23, we add 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. We also added an `@using` expression so that we can refer to the `ViewModel` classes in views without having to qualify their names with the namespace.

Listing 23. 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

```

1.29 Task 28: Adding the View Model Data

We are not quite ready to use the tag helper because we yet to provide an instance of the `PagingInfo` view model class to the view. We could do this using the view bag feature, but we would rather wrap all of the data we are going to send from the controller to the view in a single view model class. To do this, we add a class file called `ProductsListViewModel.cs` to the `Models/ViewModels` folder of the `SportsStore` project. Listing 24 shows the contents of the new file.

Listing 24. 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; }
    }
}

```

We need to 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 25. These changes pass a ProductsListViewModel object as the model data to the view.

Listing 25. Updating the List Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
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()
                }
            });
    }
}
```

The view is currently expecting a sequence of Product objects, so we need to update the List.cshtml file, as shown in Listing 26, to deal with the new view model type.

Listing 26. 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>
}
```

```
    </div>
}
```

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

1.30 Task 29: Displaying the Page Links

We have everything in place to add the page links to the `List` view. We 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 27.

Listing 27. 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 36. The style is still basic, which we will fix later. 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.

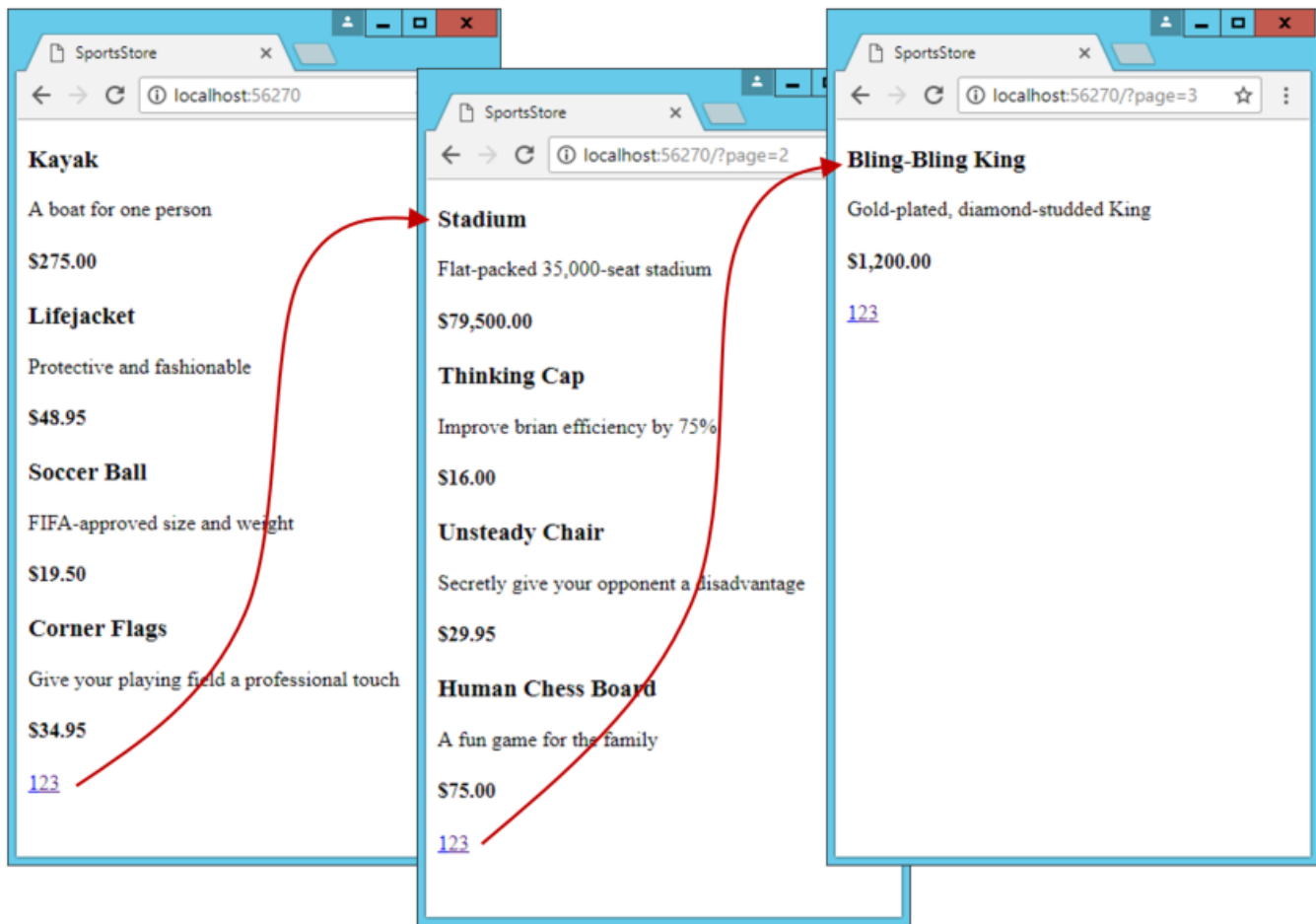


Figure 36. Displaying page navigation links

1.31 Task 30: Improving the URLs

We 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
```

We 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 we 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 28.

Listing 28. Adding a New Route in the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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();
        }

        // This method gets called by the runtime. Use this method to add services to the
        container.
        // For more information on how to configure your application, visit
        https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        pipeline.
        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?}");
            });
        }
    }
}
```

It is important that you add this route before the Default one that is already in the method. The routing system processes routes in the order they are listed, and we 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 we use to generate the page navigation links.

If you run the application and click a pagination link, you will see the new URL scheme in action, as illustrated in Figure 37.

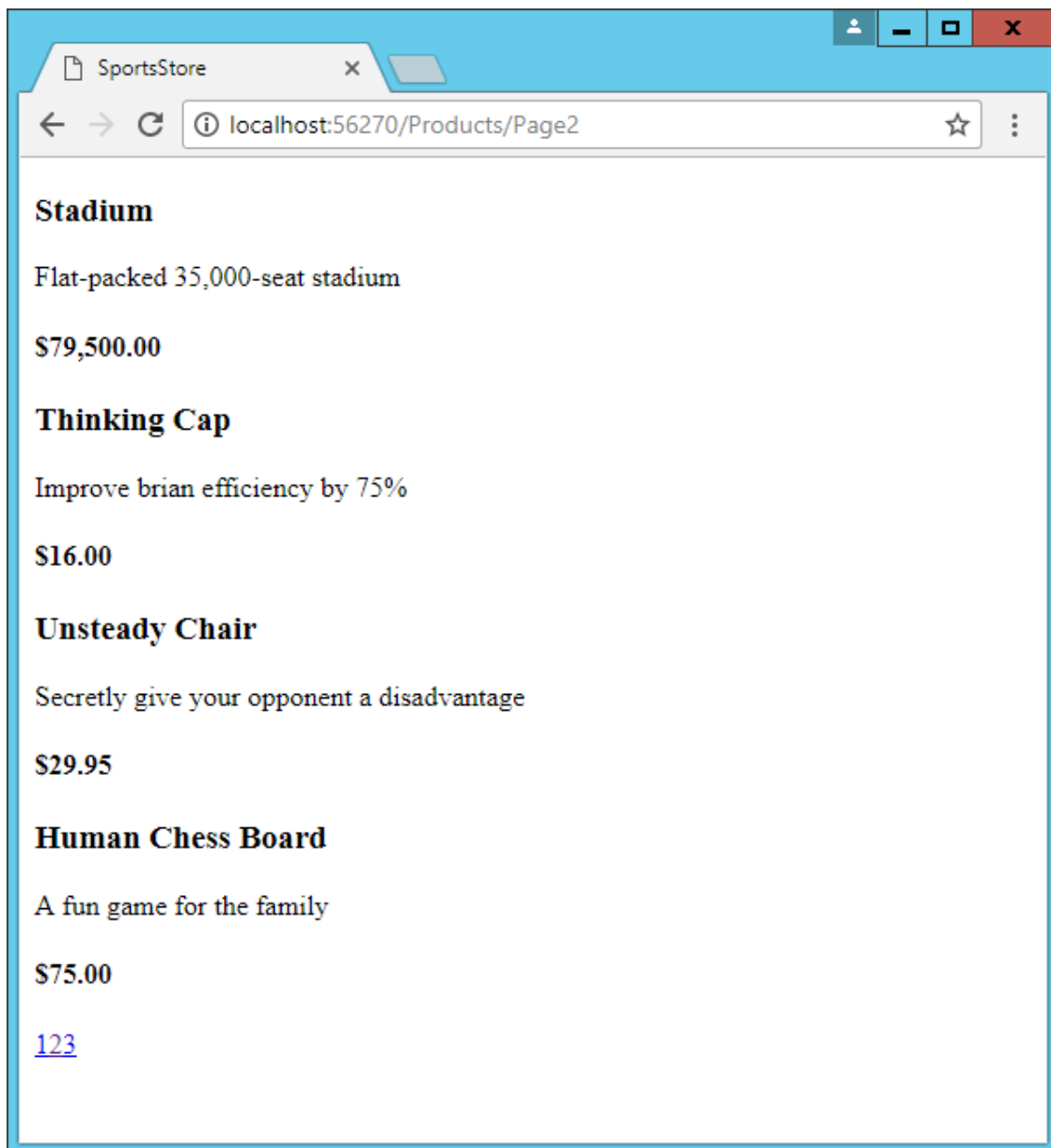


Figure 37. The new URL scheme displayed in the browser

Note: Understand URL Routing

Early versions of ASP.NET assumed that there was a direct relationship between requested URLs and the files on the server hard disk. The job of the server was to receive the request from the browser and deliver the output from the corresponding file. This approach worked just fine for Web Forms, where each ASPX page is both a file and a self-contained response to a request.

To handle MVC URLs, the ASP.NET platform uses the routing system, which has been overhauled for ASP.NET Core. Routing system lets you create any pattern of URLs you desire and express them in a clear and concise manner. The routing system has two functions as listed below. Table 15-1 puts routing into context.

- Examine an incoming URL and select the controller and action to handle the request.
- Generate outgoing URLs. These are the URLs that appear in the HTML rendered from views so that a specific action will be invoked when the user clicks the link (at which point, it becomes an incoming URL again).

Question	Answer
What are they?	The routing system is responsible for processing incoming requests and selecting controllers and action methods to process them. The routing system is also used to generate routes in views, known as outgoing URLs.
Why are they useful?	The routing system allows requests to be handled flexibly without URLs being tied to the structure of classes in the Visual Studio project.
How are they used?	The mapping between URLs and the controllers and action methods is defined in the Startup.cs file or by applying the Route attribute to controllers.
Are there any pitfalls or limitations?	The routing configuration for a complex application can become hard to manage.
Are there any alternatives?	No. The routing system is an integral part of ASP.NET Core.
Have they changed since MVC 5?	<p>The routing system works in largely the same way as with previous versions but with changes to reflect closer integration with the ASP.NET Core platform.</p> <ul style="list-style-type: none">• Convention-based routes are defined in the Startup.cs file, rather than the now-obsolete RouteConfig.cs file.• The routing classes are now defined in the Microsoft.AspNetCore.Routing namespace.• Routes no longer match URLs if there is no corresponding controller and action method in the application (in previous versions of MVC, routes could match a URL but still return a 404 – Not Found error.• Convention-based default values, optional segments, and route constraints can now be expressed as part of the URL pattern, using the same syntax as with attribute-based routing.• Requests for static files (such as images, CSS, and JavaScript) are now handled by dedicated middleware.

You can learn more about it at <https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing>.

1.32 Styling the Content

We have built a great deal of infrastructure and the basic features of the SportsStore application are starting to come together, but we have not paid any attention to appearance. In this section, we will put some of that right. We are going to implement a classic two-column layout with a header, as shown in Figure 38.

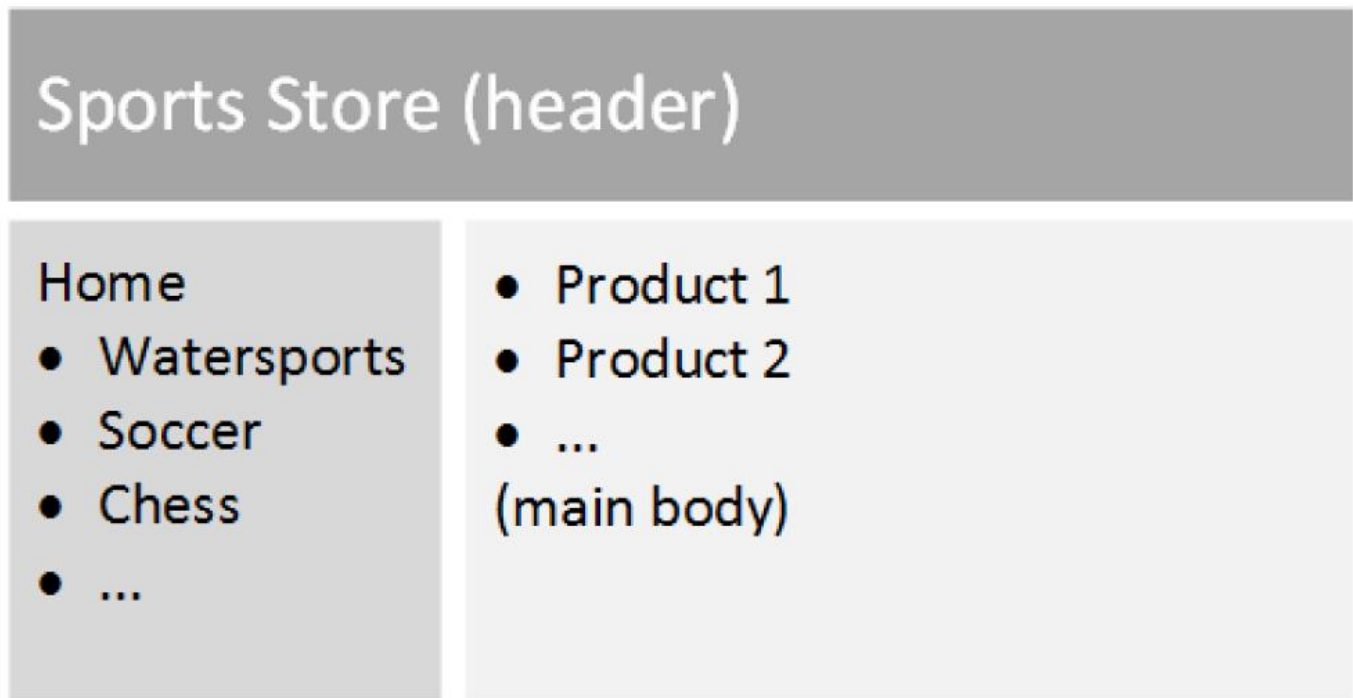


Figure 38. The design goal for the SportsStore application

1.33 Task 31: Installing the Bootstrap Package

We are going to use the Bootstrap package to apply the CSS styles to the application. We will rely on the Visual Studio support for Bower to install the Bootstrap package. A client-side package is one that contains content that is sent to the client, such as JavaScript files, CSS stylesheets or images. In previous MVC versions, NuGet was used to these projects as well, but ASP.NET Core MVC relies on a new tool, called Bower. Bower is an open-source tool that has been developed outside of Microsoft and the .NET world and is widely used in non-ASP.NET web application development. In fact, Bower has become so successful that some popular client-side packages are only distributed through Bower.

Bower packages are specified through the bower.json file. To create this bower.json file, right click the SportStore project item in the Solution Explorer, select Add ► New Item from the pop-up menu, and choose 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 Figures 39 and 40.

Visual Studio sets the name to bower.json, and clicking the Add button adds the file to the project with the default content. Listing 58 shows the addition of a client-side package to the bower.json file, which is done by adding an entry to the dependencies section using the same format as the project.json file.

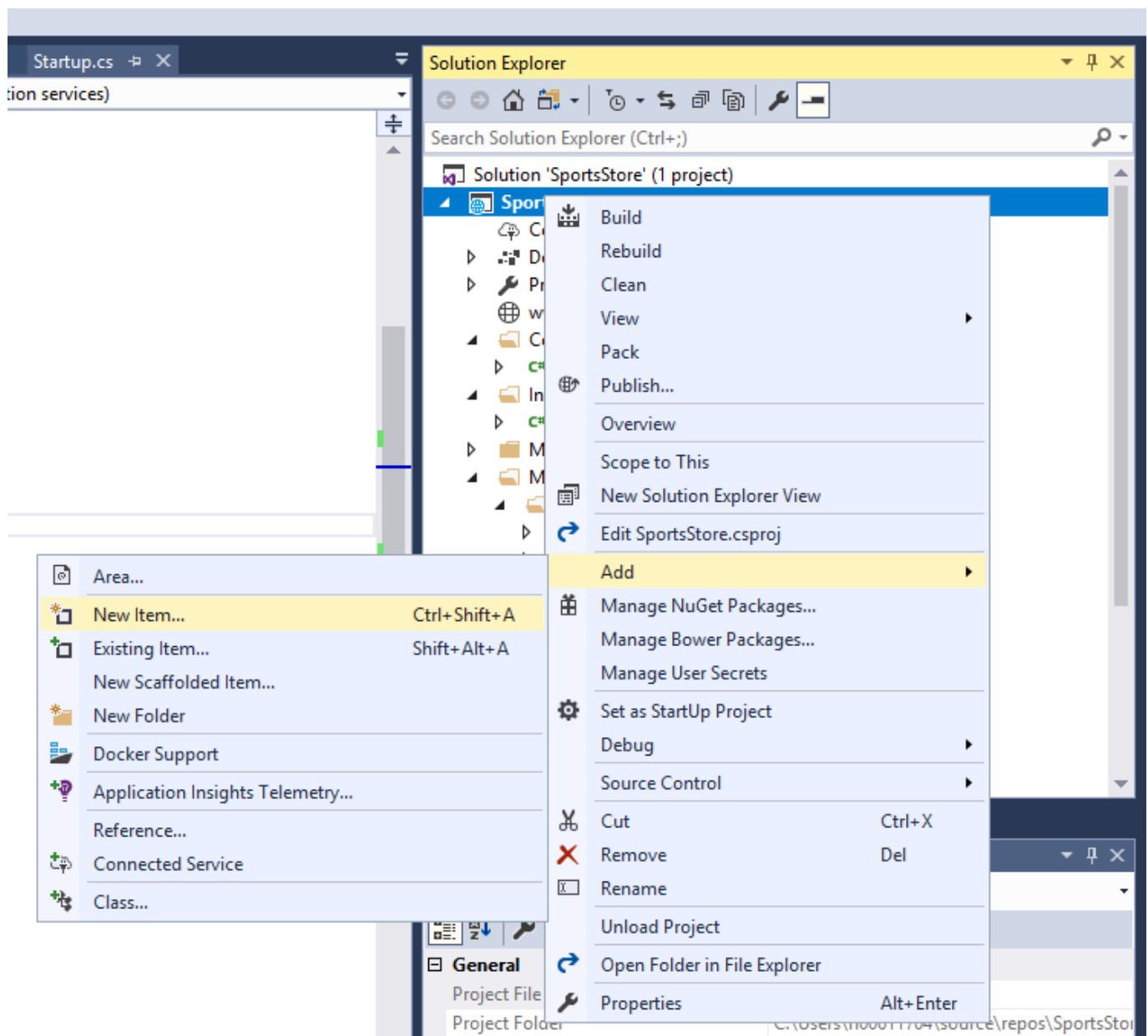


Figure 39. Adding a New Item to the Project Solution set

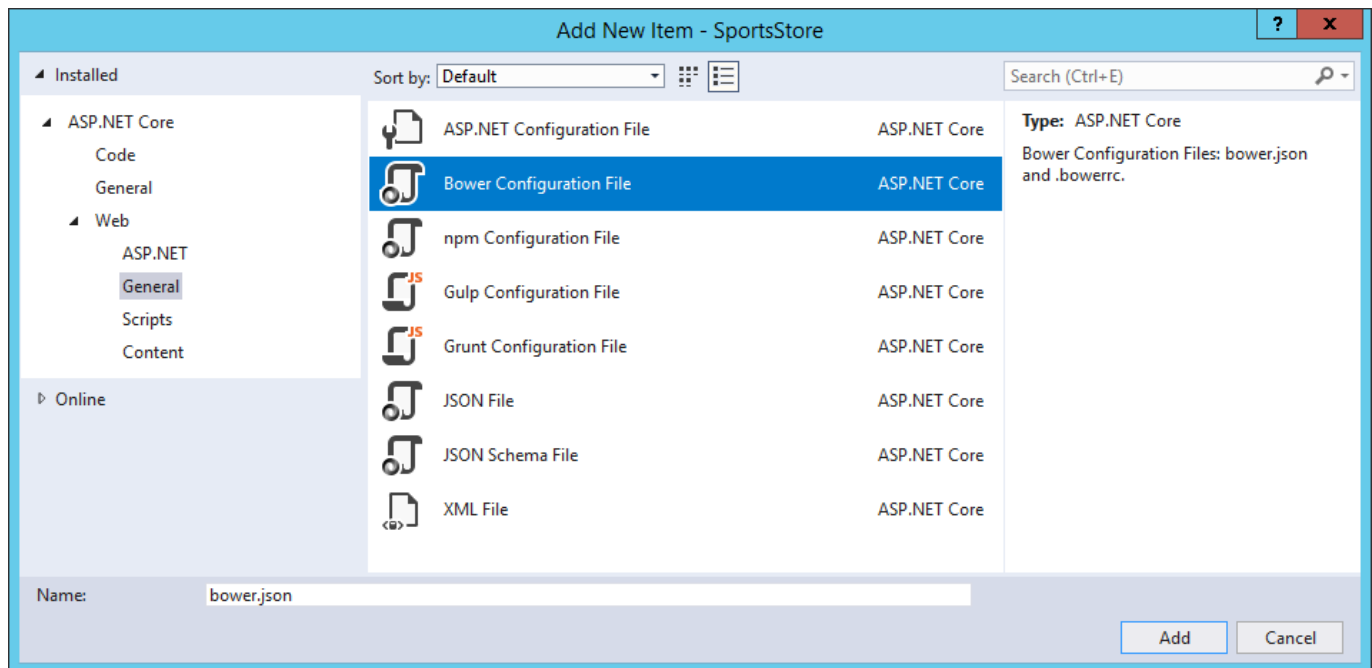


Figure 40. Creating the Bower configuration file

Listing 29. Adding Bootstrap to the bower.json File in the SportsStore Project

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "jquery": "3.3.1",
    "bootstrap": "4.0.0"
  }
}
```

The addition in the listing adds the Bootstrap CSS package to the example project. When you edit the bower.json file, Visual Studio will offer you a list of package names and list the versions of the packages that are available, as shown in Figure 41.

Version numbers can be specified in a range of different ways in the bower.json file, the most useful of which are described in Table 4. The safest way to specify a package is to use an explicit version number. This ensures that you will always be working with the same version unless you deliberately update the bower.json file to request a different one.

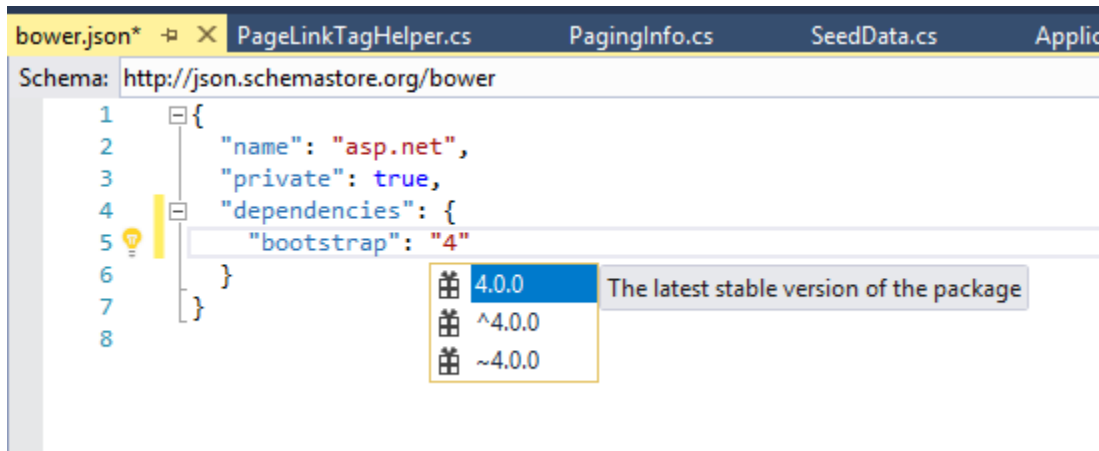


Figure 41. Listing the available versions of the client-side package

Table 4. Common Formats for Version Numbers in the bower.json File

Format	Description
3.3.6	Expressing a version number directly will install the package with the exactly matching version number, e.g., 3.3.6.
*	Using an asterisk will allow Bower to download and install any version of the package.
>3.3.6 >=3.3.6	Prefixing a version number with > or > = will allow Bower to install any version of the package that is greater than or greater than or equal to a given version.
<3.3.6 <=3.3.6	Prefixing a version number with < or < = will allow Bower to install any version of the package that is less than or less than or equal to a given version.
~3.3.6	Prefixing a version number with a tilde (the ~ character) will allow Bower to install versions even if the patch level number (the last of the three version numbers) doesn't match). For example, specifying ~3.3.6 will allow Bower to install version 3.3.7 or 3.3.8 (which would be patches to version 3.3.6) but not version 3.4.0 (which would be a new minor release).
^3.3.6	Prefixing a version number with a caret (the ^ character) will allow Bower to install versions even if the minor release number (the second of the three version numbers) or the patch number doesn't match. For example, specifying ^3.3.0 will allow Bower to install versions 3.3.1, 3.4.0, and 3.5.0, for example, but not version 4.0.0.

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 thus we have added jQuery to the project as well.

Like NuGet, Bower manages the dependencies of the packages you add to a project. Bootstrap relies on the jQuery JavaScript library for some of its advanced features, which is why there are two packages shown in the Figure 42. You can see the list of packages and their dependencies by expanding the Dependencies item in the Solution Explorer. You can learn more about Bower at <https://docs.microsoft.com/en-us/aspnet/core/client-side/bower>.

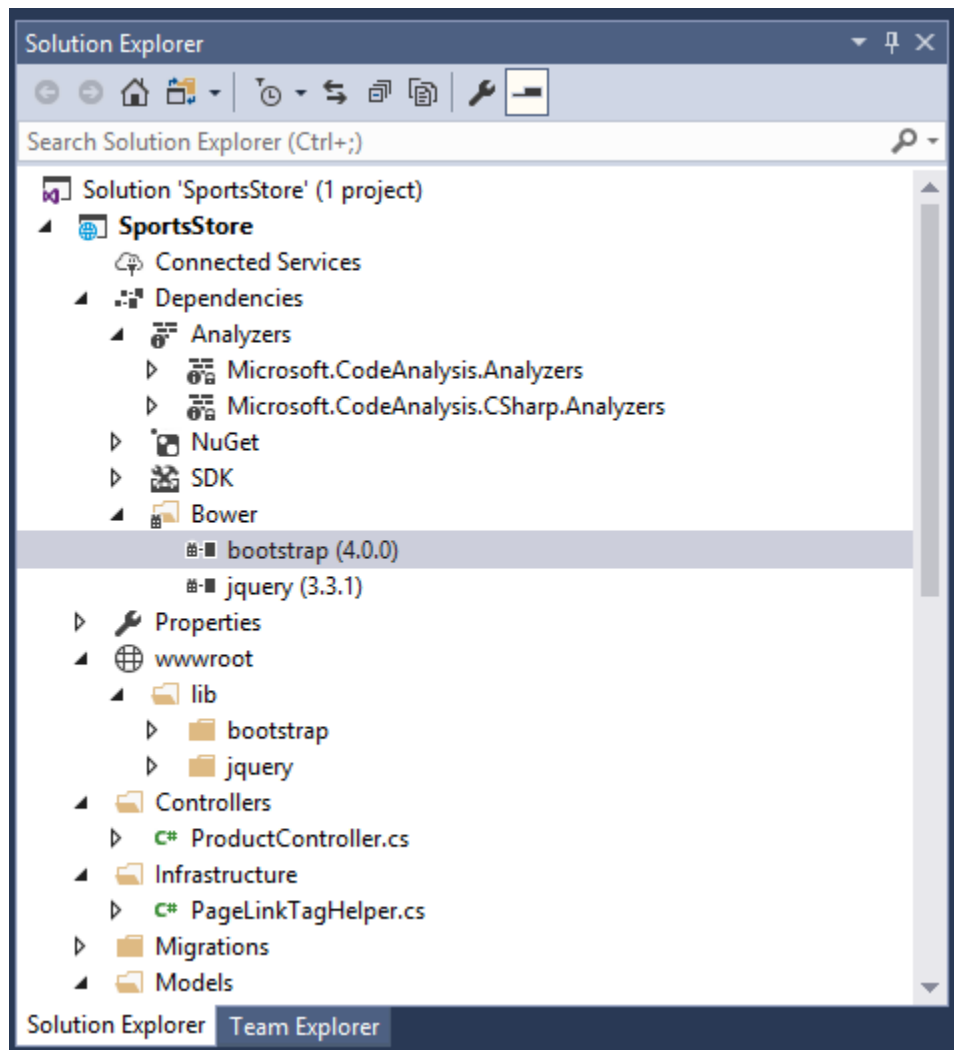


Figure 42. Examining the client-side packages and their dependencies

1.34 Task 32: Applying Bootstrap Styles to the Layout

The view start file that we added in Task 12 specified that a file called `_Layout.cshtml` in the Views/Shared Folder should be used as the default layout, and that is where the initial Bootstrap styling will be applied, as shown in Listing 30.

Listing 30. 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>
  <nav class="navbar navbar-inverse bg-info">
    <a class="navbar-brand text-white" href="#">SPORTS STORE</a>
```

```

</nav>
<div class="container">
  <div class="row">
    <div class="col col-md-3">
      Put something useful here later
    </div>
    <div class="col col-md-9">
      @RenderBody()
    </div>
  </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 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 we can use the styles it defines in any of the views that rely on the layout. In Listing 31, you can see the styling we apply to the *List.cshtml* file in the *View / Product* folder.

Listing 31. Styling Content in the *List.cshtml* File

```

@model ProductsListViewModel

@foreach (var p in Model.Products)
{
  <div class="card m-3 rounded">
    <div class="card-block m-2">
      <h3>
        <strong>@p.Name</strong>
        <span class="float-right bg-success text-white rounded m-2 p-1">
          @p.Price.ToString("c")
        </span>
      </h3>
      <span class="lead">@p.Description</span>
    </div>
  </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 float-right">
</div>

```

We need to style the buttons that are generated by the *PageLinkTagHelper* class in the *Infrastructure* folder, but we 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, we have define custom attributes on the *div* element that specify the classes that we require, and these correspond to properties we added to the tag helper class, which are then used to style the *a* elements that are produced, as shown in Listing 32.

Listing 32. Adding Classes to Generated Elements in 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 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 43.

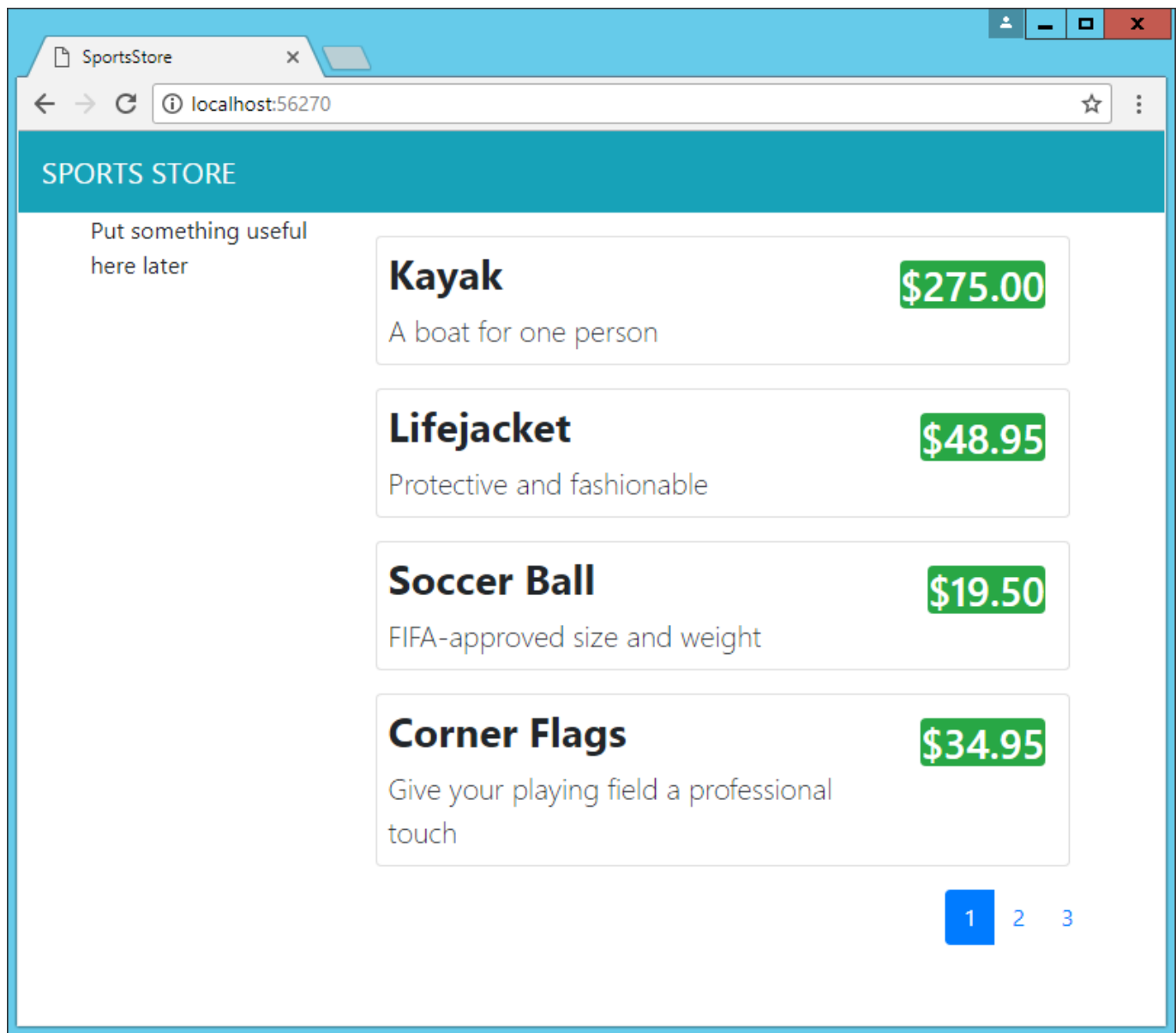


Figure 43. The design-enhanced SportsStore application

1.35 Task 33: Creating a Partial View

You will often need to use the same fragments of Razor tags and HTML markup in several different places in the application. Rather than duplicate the content, you can use partial views, which are separate view files that contain fragments of tags and markup that can be included in other views. Partial views are just regular CSHTML files, and it is their use that differentiates them from regular Razor views. Visual Studio provides some tooling support for creating prepopulated partial views, but the simplest way to create a partial view is to create a regular view using the MVC View Page item template.

For the purposes of partial view demonstration, we are going to refactor the application to simplify the List.cshtml view. We are 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 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, we add a Razor view file called ProductSummary.cshtml to the Views/Shared folder and added the markup shown in Listing 33.

Listing 33. The Contents of the ProductSummary.cshtml File in the Views/Shared Folder

```
@model Product
<div class="card m-3 rounded">
  <div class="card-block m-2">
    <h3>
      <strong>@Model.Name</strong>
      <span class="float-right bg-success text-white rounded m-2 p-1">
        @Model.Price.ToString("c")
      </span>
    </h3>
    <span class="lead">@Model.Description</span>
  </div>
</div>
```

A partial view is consumed by calling the @Html.Partial expression from within another view. We need to update the List.cshtml file in the Views/Products folder so that it uses the partial view, as shown in Listing 34.

Listing 34. 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 float-right">
</div>
```

We have taken the markup that was previously in the foreach loop in the List.cshtml view and moved it to the new partial view. We 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. If you run the application, you will note that 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.

1.36 Adding Navigation Controls

The SportsStore application will be more usable if customers can navigate products by category. We will do this in three phases.

- Enhance the List action model in the ProductController class so that it is able to filter the Product objects in the repository.
- Revisit and enhance the URL scheme.
- Create a category list that will go into the sidebar of the site, highlighting the current category and linking to others.

1.37 Task 34: Filtering the Product List

We are going to start by enhancing the view model class, ProductsListViewModel. We need to communicate the current category to the view in order to render the sidebar, and this is as good a place to start as any. Listing 35 shows the changes we need to make in the ProductsListViewModel.cs file in the Models/ViewModels folder.

Listing 35. Adding a Property in the ProductsListViewModel.cs File

```
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; }

        public string CurrentCategory { get; set; }
    }
}
```

We added a property called CurrentCategory. The next step is to update the Product controller so that the List action method will filter Product objects by category and use the new property we added to the view model to indicate which category has been selected. Listing 36 shows the changes.

Listing 36. Adding Category Support to the List Action Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
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 ViewResult List(string category, int page = 1)
=> View(new ProductsListViewModel {
    Products = repository.Products
        .Where(p => category == null || p.Category == category)
        .OrderBy(p => p.ProductID)
        .Skip((page - 1) * PageSize)
        .Take(PageSize),
    PagingInfo = new PagingInfo {
        CurrentPage = page,
        ItemsPerPage = PageSize,
        TotalItems = repository.Products.Count()
    },
    CurrentCategory = category
});
}
}

```

We made three changes to the action method. First, we added a parameter called `category`. This category parameter is used by the second change in the listing, which is an enhancement to the LINQ query: if `category` is not null, only those `Product` objects with a matching `Category` property are selected. The last change is to set the value of the `CurrentCategory` property we added to the `ProductsListViewModel` class. However, these changes mean that the value of `PagingInfo.TotalItems` is incorrectly calculated because it doesn't take the category filter into account. We will fix this later.

To see the effect of the category filtering, start the application and select a category using the following query string, changing the port to match the one that Visual Studio assigned for your project (and taking care to use an uppercase S for Soccer):

`http://localhost:55627/?category=Soccer`

You will see only the products in the Soccer category, as shown in Figure 44. Obviously, users won't want to navigate to categories using URLs, but you can see how small changes can have a big impact in an MVC application once the basic structure is in place.

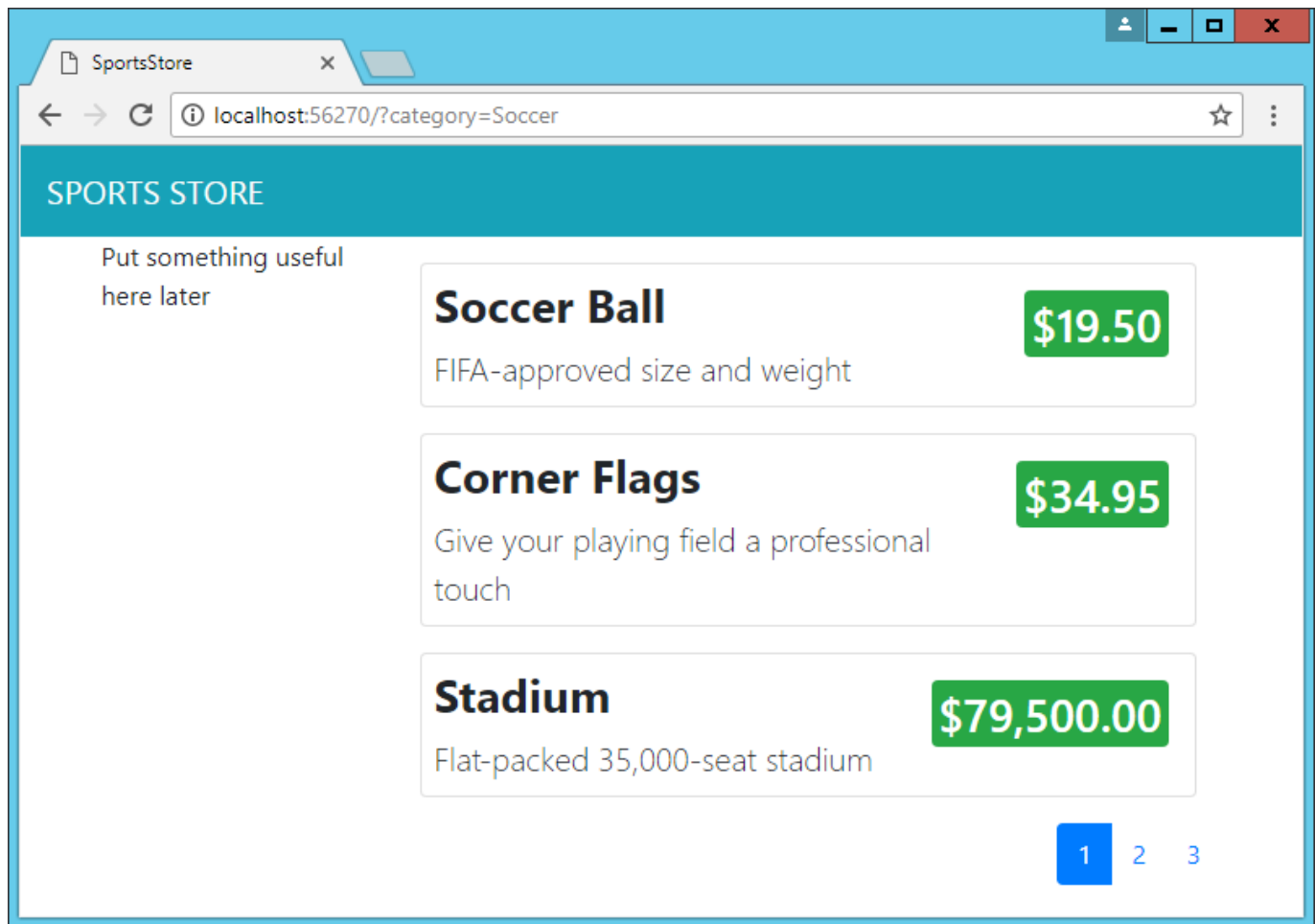


Figure 44. Using the query string to filter by category

1.38 Task 35: Refining the URL Scheme

No one wants to see or use ugly URLs such as `/?category=Soccer`. To address this, we are going to change the routing configuration in the `Configure` method of the `Startup` class to create a more useful set of URLs, as shown in Listing 37.

Listing 37. Changing the Routing Schema in the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
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();
    }

    // This method gets called by the runtime. Use this method to add services to the
    container.
    // For more information on how to configure your application, visit
    https://go.microsoft.com/fwlink/?LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
        services.AddTransient<IProductRepository, EFProductRepository>();

        services.AddMvc();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request
    pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
        app.UseStaticFiles();
        app.UseMvc(routes => {

            routes.MapRoute(
                name: null,
                template: "{category}/Page{page:int}",
                defaults: new { controller = "Product", action = "List" }
            );
            routes.MapRoute(
                name: null,
                template: "Page{page:int}",
                defaults: new { controller = "Product", action = "List", page = 1 }
            );
            routes.MapRoute(
                name: null,
                template: "{category}",
                defaults: new { controller = "Product", action = "List", page = 1 }
            );
            routes.MapRoute(
                name: null,
                template: "",
                defaults: new { controller = "Product", action = "List", page = 1 });

            routes.MapRoute(name: null, template: "{controller}/{action}/{id?}");

        });
    }
}

```

Table 5. Route Summary

URL	Leads To
/	Lists the first page of products from all categories
/Page2	Lists the specified page (in this case, page 2), showing items from all categories.
/Soccer	Shows the first page of items from a specific category (in this case, the Soccer category)
/Soccer/Page2	Shows the specified page (in this case, page 2) of items from the specified category (in this case, Soccer)

Table 5 describes the URL scheme that these routes represent. The ASP.NET Core routing system is used by MVC to handle incoming requests from clients, but it also generates outgoing URLs that conform to the URL scheme and that can be embedded in web pages. By using the routing system both to handle incoming requests and to generate outgoing URLs, we can ensure that all the URLs in the application are consistent.

The `IUrlHelper` interface provides access to the URL-generating functionality. We used this interface and the `Action` method it defines in the tag helper. Now that we want to start generating more complex URLs, we need a way to receive additional information from the view without having to add extra properties to the tag helper class. Fortunately, tag helpers have a nice feature that allows properties with a common prefix to be received all together in a single collection, as shown in Listing 67.

Listing 38. Receiving Prefixed Attribute Values 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;
using System.Collections.Generic;

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; }

        [HtmlAttributeName(DictionaryAttributePrefix = "page-url-")]
        public Dictionary<string, object> PageUrlValues { get; set; } = new Dictionary<string,
object>();

        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");
        PageUrlValues["page"] = i;
        tag.Attributes["href"] = urlHelper.Action(PageAction, PageUrlValues);

        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);
}
}
}

```

Decorating a tag helper property with the *HtmlAttributeName* attribute allows us to specify a prefix for attribute names on the element, which in this case will be *page-url-*. The value of any attribute whose name begins with this prefix will be added to the dictionary that is assigned to the *PageUrlValues* property, which is then passed to the *IUrlHelper.Action* method to generate the URL for the *href* attribute of the *a* elements that the tag helper produces.

In the *List.cshtml* file, we add a new attribute to the *div* element that is processed by the tag helper, specifying the category that will be used to generate the URL, as shown in Listing 39. We add only one new attribute to the view, but any attribute with the same prefix would be added to the dictionary.

Listing 39. Adding a New Attribute 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" page-url-category="@Model.CurrentCategory" class="btn-
group float-right">
</div>

```

Prior to this change, the links generated for the pagination links were like this:

```
http://<myserver>:<port>/Page1
```

If the user clicked a page link like this, the category filter would be lost, and the application would present a page containing products from all categories. By adding the current category, taken from the view model, we generate URLs like this instead:

```
http://<myserver>:<port>/Chess/Page1
```

When the user clicks this kind of link, the current category will be passed to the List action method, and the filtering will be preserved. After you have made this change, you can visit a URL such as /Chess or /Soccer, and you will see that the page link at the bottom of the page correctly includes the category, as shown in the Figure 45.

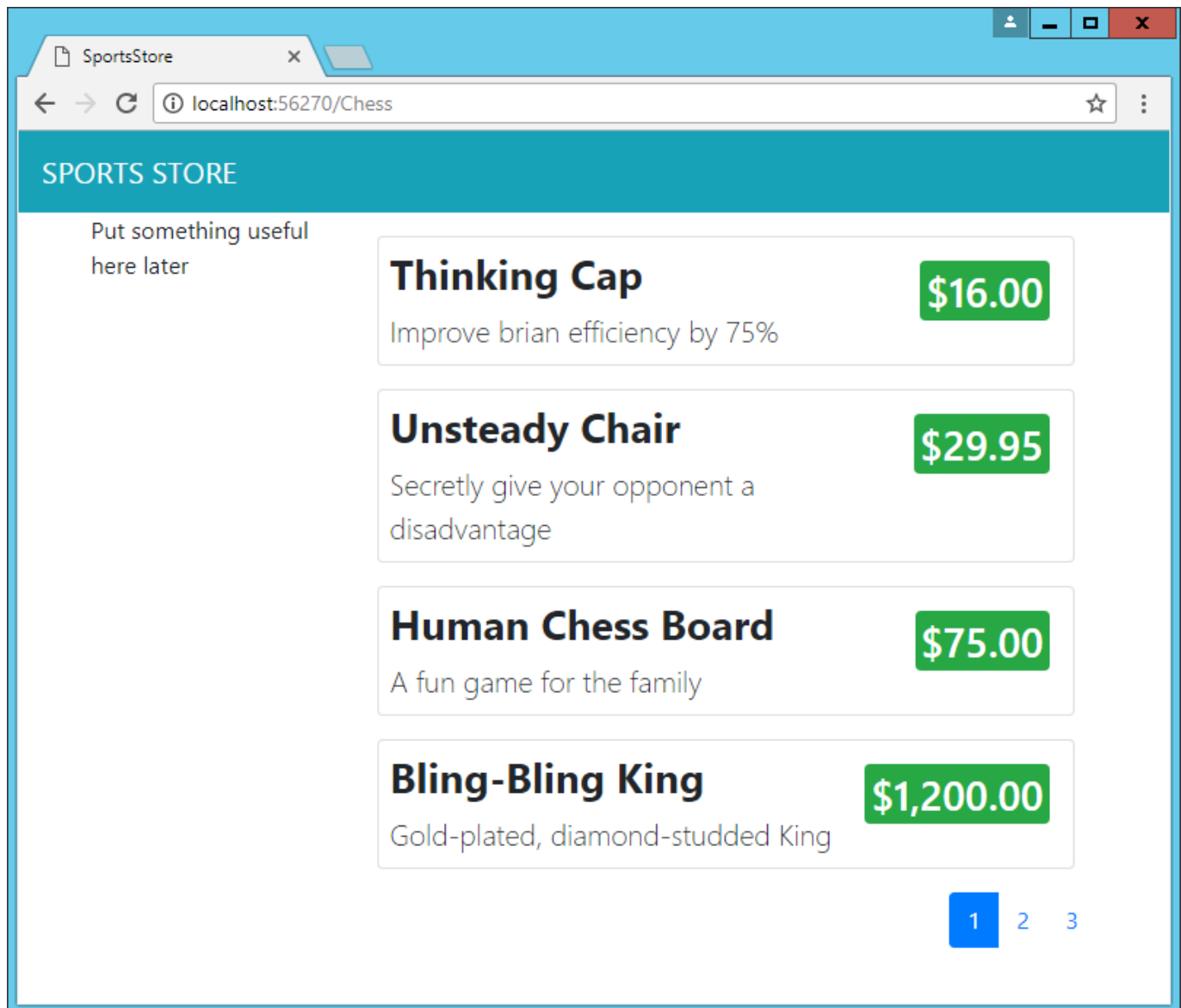


Figure 45. Category Refined URL view

1.39 Task 36: Building a Category Navigation Menu

We need to provide customers with a way to select a category that does not involve typing in URLs. This means presenting them with a list of the categories available and indicating which, if any, is currently selected. As we build out the application, we will use this list of categories in more than one controller, so we need something that is self-contained and reusable.

ASP.NET Core MVC has the concept of view components, which are perfect for creating items such as a reusable navigation control. A view component is a C# class that provides a small amount of reusable application logic with the ability to select and display Razor partial views.

In this case, we will create a view component that renders the navigation menu and integrates it into the application by invoking the component from the shared layout. This approach gives us a regular C# class that can contain whatever application logic we need. It is a nice way of creating smaller segments of an application while preserving the overall MVC approach.

Creating the Navigation View Component

We create a folder called Components (see Figure 46), which is the conventional home of view components, and add to it a class called `NavigationMenuViewComponent.cs`, which we use to define the class shown in Listing 40.

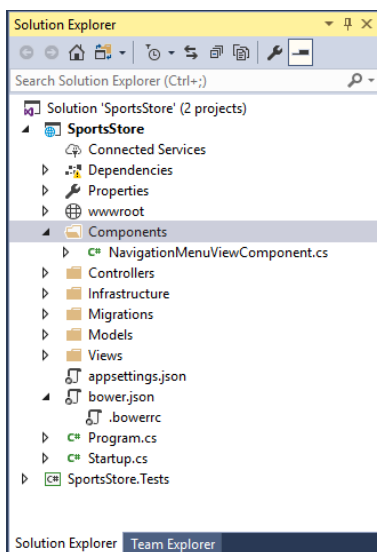


Figure 46. Components Folder in the Project Solution set

Note: Understanding View Components

View components are a new addition in ASP.NET Core MVC and replace the child action feature from previous versions. View components are classes that provide action-style logic to support partial views, which means complex content to be embedded in views while allowing the C# code that supports it to be easily maintained. Below table puts view components in context.

Applications commonly need to embed content in views that isn't related to the main purpose of the application. Common examples include site navigation tools, tag clouds, and authentication panels that let the user log in without visiting a separate page.

The common thread that all these examples have is that the data required to display the embedded content isn't part of the model data passed from the action to the view.

Partial views are used to create reusable markup that is required in views, avoiding the need to duplicate the same content in multiple places in the application. Partial views are a useful feature, but they just contain fragments of HTML and Razor directives, and the data they operate on is received from the parent view. If you need to display different data, then you run into a problem. You could access the data you need directly from the partial view, but this breaks the separation of concerns that underpins the MVC pattern and results in data retrieval and processing logic being placed in a view file. Alternatively, you could extend the view models used by the application so that it includes the data you require, but this means you have to change every action method and makes it hard to isolate the functionality of action methods.

This is where view components come in. A view component is a C# class that provides a partial view with the data that it needs, independently from the parent view and the action that renders it. In this regard, a view component can be thought of as a specialized action, but one that is used only to provide a partial view with data; it cannot receive HTTP requests, and the content that it provides will always be included in the parent view.

Question	Answer
What are they?	View components are classes that provide application logic to support partial views or to inject small fragments of HTML or JSON data into a parent view.
Why are they useful?	Without view components, it is hard to create embedded functionality such as shopping baskets or login panels in a way that is easy to maintain and unit test.
How are they used?	View components are typically derived from the <code>ViewComponent</code> class and are applied in a parent view using the <code>@await Component.InvokeAsync</code> expression.
Are there any pitfalls or limitations?	No, view components are a simple and predictable feature. The main pitfall is not using them and trying to include application logic within views where it is difficult to test and maintain.
Are there any alternatives?	You could put the data access and processing logic directly in a partial view, but the result is difficult to work with and hard to test effectively.
Have they changed since MVC 5?	View components are a new feature in ASP.NET Core MVC, replacing the child actions feature from previous versions.

Listing 40. The Contents of the `NavigationMenuViewComponent.cs` File in the Components Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Components
{
    public class NavigationMenuViewComponent : ViewComponent
```

```

{
    public string Invoke()
    {
        return "Hello from the Nav View Component";
    }
}

```

The view component's *Invoke* method is called when the component is used in a Razor view, and the result of the *Invoke* method is inserted into the HTML sent to the browser. We started with a simple view component that returns a string, but we will replace this with dynamic HTML content shortly. We want the category list to appear on all pages, so we are going to use the view component in the shared layout, rather than in a specific view. Within a view, view components are used through the `@await Component.InvokeAsync` expression, as shown in Listing 41.

Listing 41. Using View Component in the `_Layout.cshtml` File

```

<!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>
    <nav class="navbar navbar-inverse bg-info">
        <a class="navbar-brand text-white" href="#">SPORTS STORE</a>
    </nav>
    <div class="container">
        <div class="row">
            <div class="col col-md-3">
                @await Component.InvokeAsync("NavigationMenu")
            </div>
            <div class="col col-md-9">
                @RenderBody()
            </div>
        </div>
    </div>
</body>
</html>

```

We removed the placeholder text and replaced it with a call to the `Component.InvokeAsync` method. The argument to this method is the name of the component class, omitting the `ViewComponent` part of the class name, such that `NavigationMenu` specifies the `NavigationMenuViewComponent` class. If you run the application, you will see that the output from the `Invoke` method is included in the HTML sent to the browser, as shown in Figure 47.

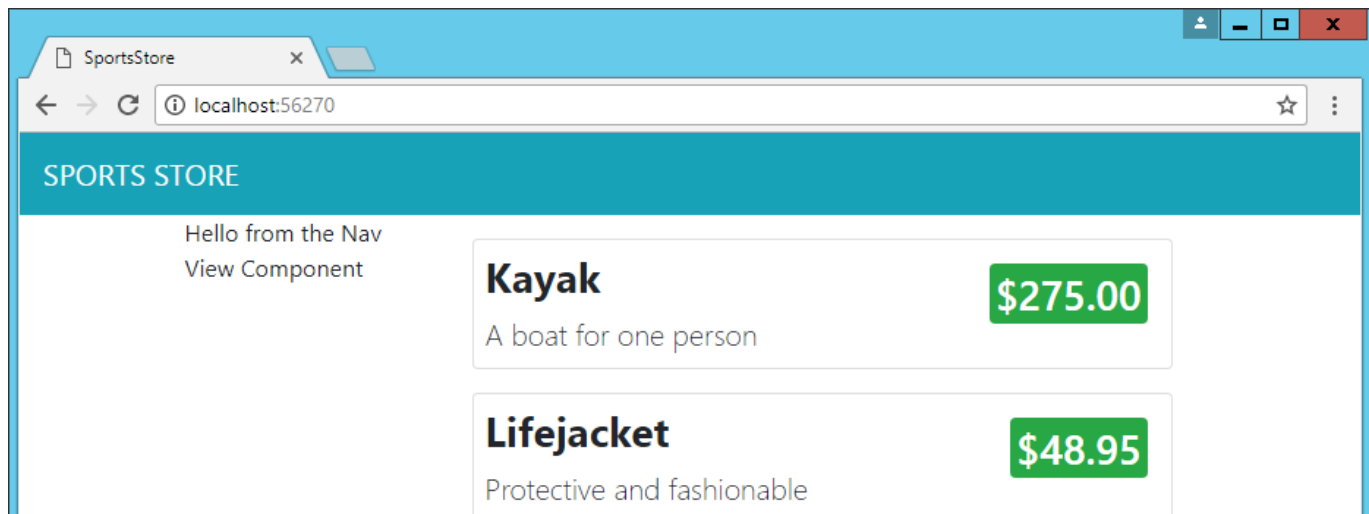


Figure 47. Using a view component

Generating Category Lists

We can now return to the navigation view controller and generate a real set of categories. We could build the HTML for the categories programmatically, as we did for the page tag helper, but one of the benefits of working with view components is they can render Razor partial views. That means we can use the view component to generate the list of components and then use the more expressive Razor syntax to render the HTML that will display them. The first step is to update the view component, as shown in Listing 42.

Listing 42. Adding the Categories List in the NavigationMenuViewComponent.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using SportsStore.Models;

namespace SportsStore.Components
{
    public class NavigationMenuViewComponent : ViewComponent
    {
        private IProductRepository repository;

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

        public IViewComponentResult Invoke()
        {
            return View(repository.Products
                .Select(x => x.Category)
                .Distinct()
                .OrderBy(x => x));
        }
    }
}
```

```

    }
}
}

```

The constructor defined in Listing 42 defines an *IProductRepository* argument. When MVC needs to create an instance of the view component class, it will note the need to provide this argument and inspect the configuration in the Startup class to determine which implementation object should be used. This is the same dependency injection feature that we used in the *ProductController*, and it has the same effect, which is to allow the view component to access data without knowing which repository implementation will be used.

In the Invoke method, we use LINQ to select and order the set of categories in the repository and pass them as the argument to the View method, which renders the default Razor partial view, details of which are returned from the method using an *ViewComponentResult* object.

Creating the View

Razor uses different conventions for dealing with views that are selected by view components. Both the default name of the view and the locations that are searched for the view are different from those used for controllers. To that end, we create the Views/Shared/Components/NavigationMenu folder (see Figure 48) and added to it a view file called *Default.cshtml*, to which we add the content shown in Listing 43.

This view uses one of the built-in tag helpers, to create *a* elements whose *href* attribute contains a *URL* that selects a different product category. You can see the category links if you run the application, as shown in Figure 49. If you click a category, the list of items is updated to show only items from the selected category.

Listing 43. Contents of the Default.cshtml File in the Views/Shared/Components/NavigationMenu Folder

```

@model IEnumerable<string>

<a class="btn btn-outline-dark btn-block m-2"
    asp-action="List"
    asp-controller="Product"
    asp-route-category="">
    Home
</a>

@foreach (string category in Model)
{
    <a class="btn btn-outline-dark btn-block m-2"
        asp-action="List"
        asp-controller="Product"
        asp-route-category="@category"
        asp-route-page="1">
        @category
    </a>
}

```

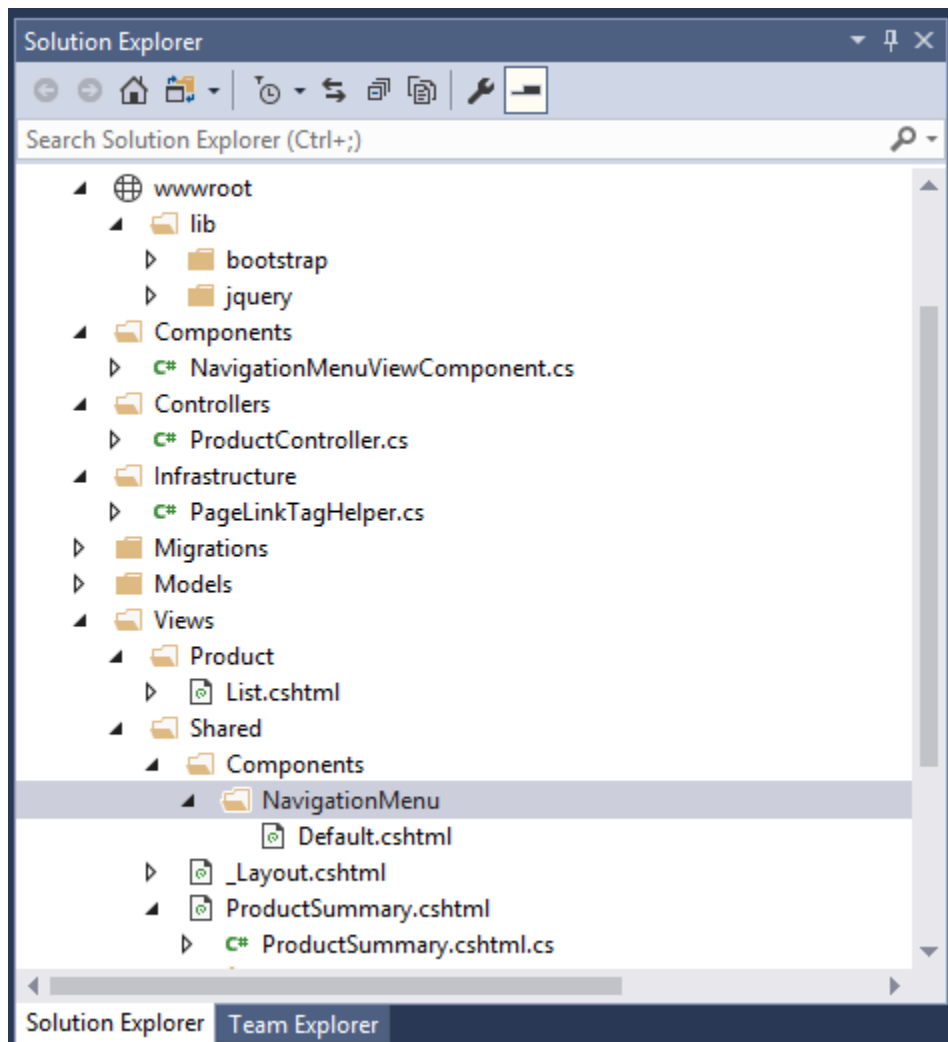


Figure 48. Views/Shared/Components/NavigationMenu Folder in the Project Solution set

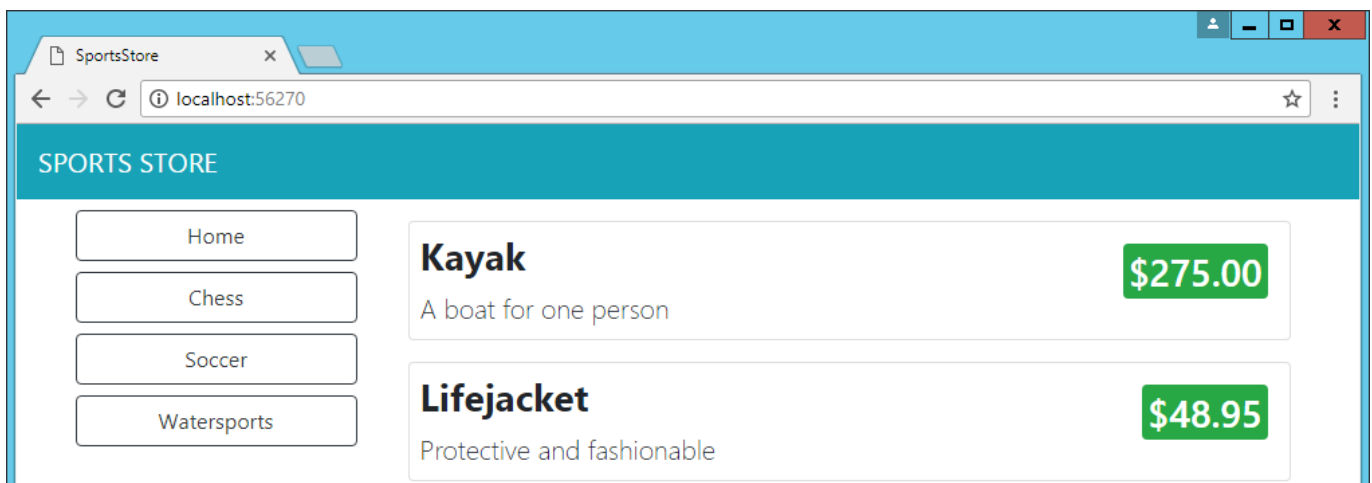


Figure 49. Generating category links with a view component

Highlighting the Current Category

There is no feedback to the user to indicate which category has been selected. It might be possible to infer the category from the items in the list, but some solid visual feedback seems like a good idea. ASP.NET Core MVC components such as controllers and view components can receive information about the current request by asking for a context object. Most of the time, you can rely on the base classes that you use to create components to take care of getting the context object for you, such as when you use the *Controller* base class to create controllers.

The *ViewComponent* base class is no exception and provides access to context objects through a set of properties. One of the properties is called *RouteData*, which provides information about how the request URL was handled by the routing system.

In Listing 44, we use the *RouteData* property to access the request data in order to get the value for the currently selected category. We could pass the category to the view by creating another view model class (and that's what we would do in a real project), but for variety, we are going to use the view bag feature. *ViewBag* is a dynamic object to which you can define new properties simply by assigning values to them, making those values available in whatever view is subsequently rendered.

Listing 44. Passing the Selected Category in the NavigationMenuViewComponent.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using SportsStore.Models;

namespace SportsStore.Components
{
    public class NavigationMenuViewComponent : ViewComponent
    {
        private IProductRepository repository;

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

        public IViewComponentResult Invoke()
        {
            ViewBag.SelectedCategory = RouteData?.Values["category"];
            return View(repository.Products
                .Select(x => x.Category)
                .Distinct()
                .OrderBy(x => x));
        }
    }
}
```

Inside the *Invoke* method, we dynamically assign a *SelectedCategory* property to the *ViewBag* object and set its value to be the current category, which is obtained through the context object returned by the *RouteData* property.

Now that we are providing information about which category is selected, we can update the view selected by the view component to take advantage of this and vary the CSS classes used to style the links to make the one representing the current category distinct from the others. Listing 74 shows the change we need to make to the Default.cshtml file.

Listing 45. Highlighting the Current Category in the Default.cshtml File

```
@model IEnumerable<string>

<a class=" btn btn-outline-dark btn-block m-2"
  asp-action="List"
  asp-controller="Product"
  asp-route-category="">
  Home
</a>

@foreach (string category in Model)
{
  <a class="btn btn-block
    @(category == ViewBag.SelectedCategory ? "btn-primary": "btn-default")"
    asp-action="List"
    asp-controller="Product"
    asp-route-category="@category"
    asp-route-page="1">
    @category
  </a>
}
```

We used a Razor expression within the class attribute to apply the *btn-primary* class to the element that represents the selected category and the *btn-default* class otherwise. These classes apply different Bootstrap styles and make the active button obvious, as shown in Figure 50.

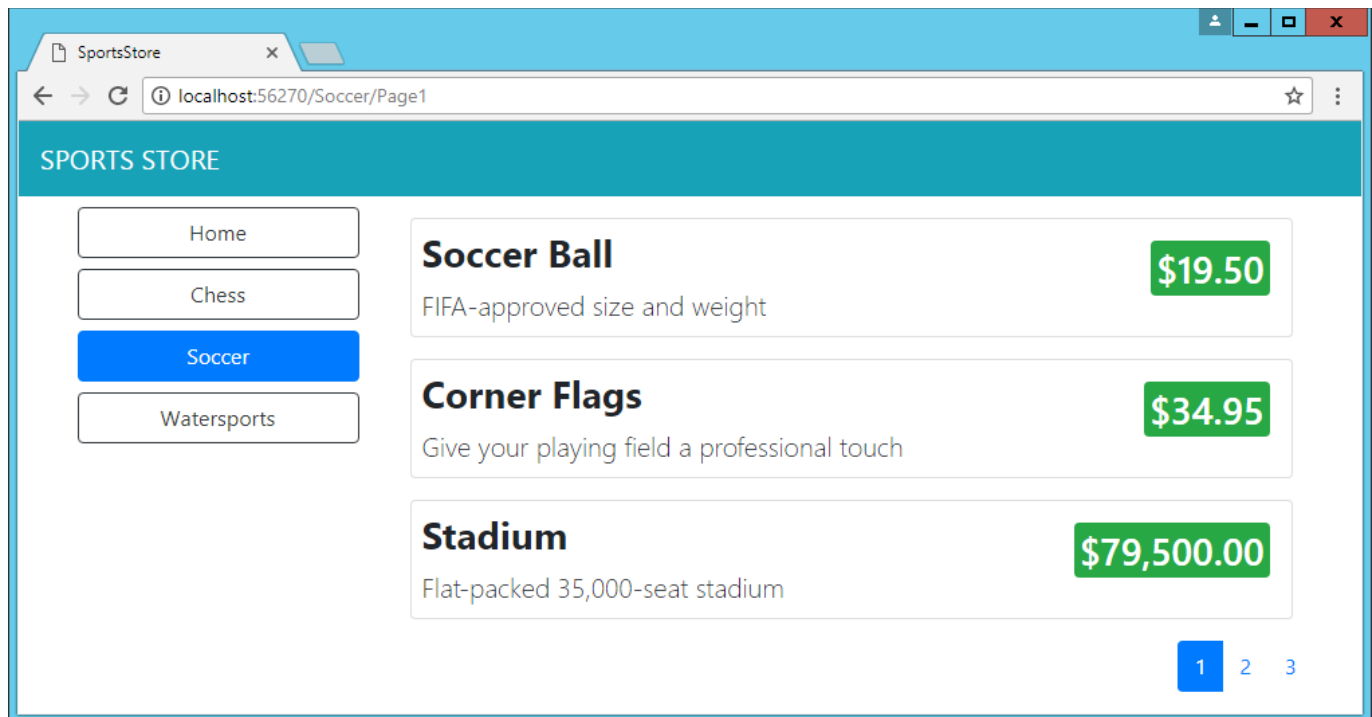


Figure 50. Highlighting the selected category

Correcting the Page Count

We need to correct the page links so that they work correctly when a category is selected. Currently, the number of page links is determined by the total number of products in the repository and not the number of products in the selected category. This means that the customer can click the link for page 2 of the Chess category and end up with an empty page because there are not enough *chess* products to fill two pages. You can see the problem in Figure 51.

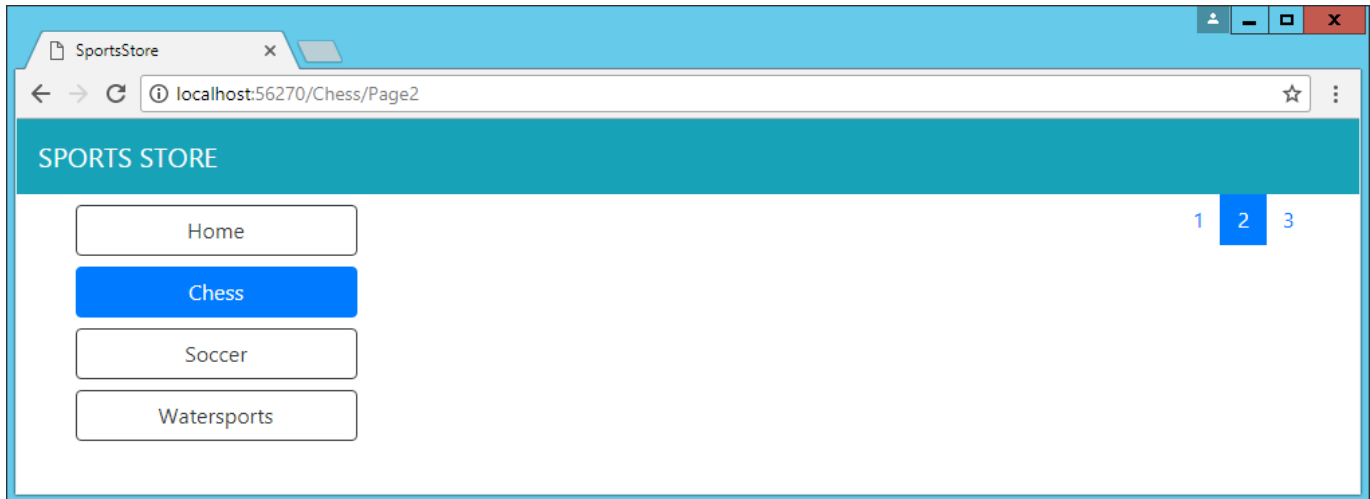


Figure 51. Displaying the wrong page links when a category is selected

We can fix this by updating the List action method in the Product controller so that the pagination information takes the categories into account, as shown in Listing 46.

Listing 46. Creating Category-Aware Pagination Data in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
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(string category, int page = 1)
            => View(new ProductsListViewModel {
                Products = repository.Products
```

```

        .Where(p => category == null || p.Category == category)
        .OrderBy(p => p.ProductID)
        .Skip((page - 1) * PageSize)
        .Take(PageSize),
        PagingInfo = new PagingInfo {
            CurrentPage = page,
            ItemsPerPage = PageSize,
            TotalItems = category == null ?
                repository.Products.Count() :
                repository.Products.Where(e =>
                    e.Category == category).Count()
        },
        CurrentCategory = category
    });
}
}

```

If a category has been selected, we return the number of items in that category; if not, we return the total number of products. Now when we view a category, the links at the bottom of the page correctly reflect the number of products in the category, as shown in Figure 52.

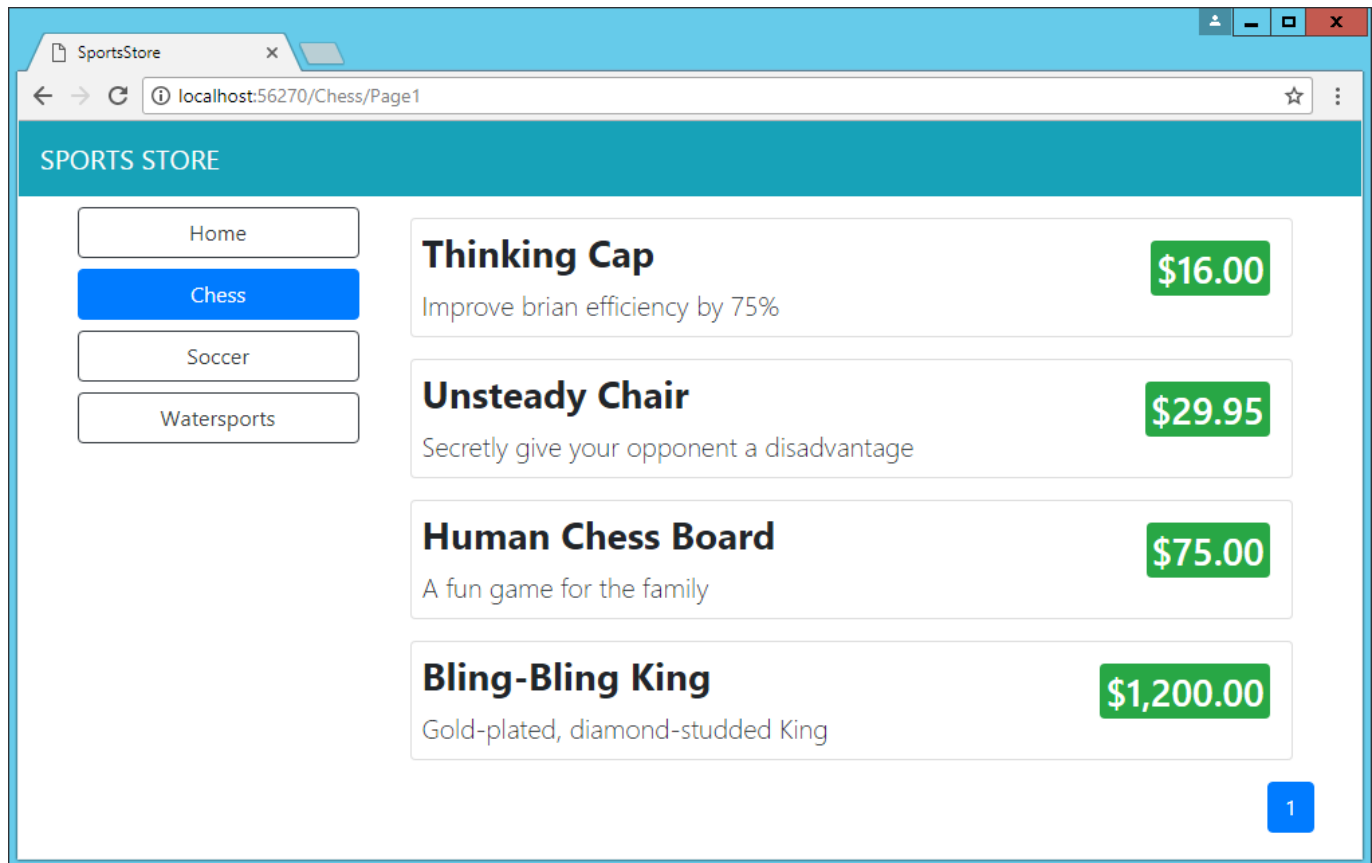


Figure 52. Displaying category-specific page counts

1.40 Task 37: Building the Shopping Cart

While the application is progressing nicely, we cannot sell any products until we implement a shopping cart. We need to create the shopping cart experience shown in Figure 53. This will be familiar to anyone who has ever made a purchase online.

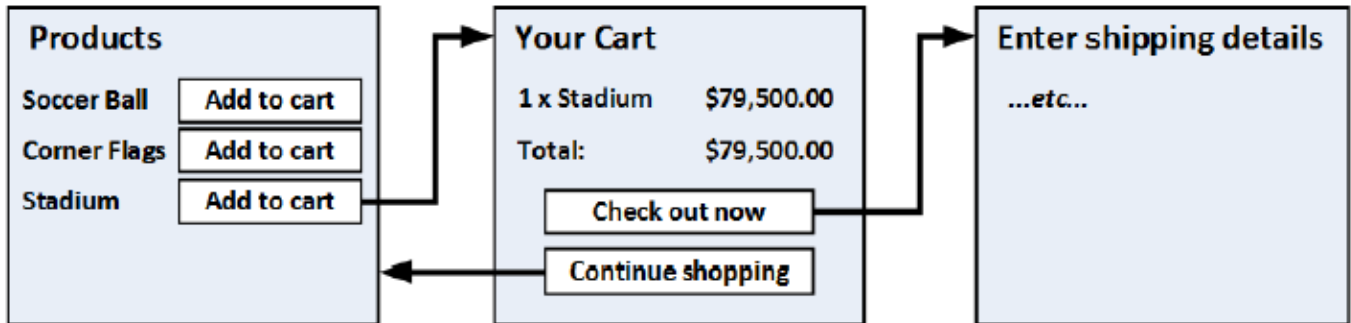


Figure 53. The basic shopping cart flow

An Add to Cart button will be displayed alongside each of the products in the catalog. Clicking this button will show a summary of the products the customer has selected so far, including the total cost. At this point, the user can click the Continue Shopping button to return to the product catalog or click the Checkout Now button to complete the order and finish the shopping session.

Defining the Cart Model

We start by adding a class file called Cart.cs to the Models folder in and used it to define the classes shown in Listing 47.

Listing 47. The Contents of the Cart.cs File in the Models Folder

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

namespace SportsStore.Models
{
    public class Cart
    {
        private List<CartLine> lineCollection = new List<CartLine>();

        public virtual void AddItem(Product product, int quantity)
        {
            CartLine line = lineCollection
                .Where(p => p.Product.ProductID == product.ProductID)
                .FirstOrDefault();

            if (line == null)
            {
                lineCollection.Add(new CartLine
                {
                    Product = product,
                    Quantity = quantity
                });
            }
        }
    }
}
```

```

        }
        else
        {
            line.Quantity += quantity;
        }
    }

    public virtual void RemoveLine(Product product) =>
        lineCollection.RemoveAll(l => l.Product.ProductID == product.ProductID);

    public virtual decimal ComputeTotalValue() =>
        lineCollection.Sum(e => e.Product.Price * e.Quantity);

    public virtual void Clear() => lineCollection.Clear();

    public virtual IEnumerable<CartLine> Lines => lineCollection;
}

public class CartLine
{
    public int CartLineID { get; set; }
    public Product Product { get; set; }
    public int Quantity { get; set; }
}
}

```

The *Cart* class uses the *CartLine* class, defined in the same file, to represent a product selected by the customer and the quantity the user wants to buy. We defined methods to add an item to the cart, remove a previously added item from the cart, calculate the total cost of the items in the cart, and reset the cart by removing all the items. We also provided a property that gives access to the contents of the cart using an *IEnumerable<CartLine>*. This is all straightforward stuff, easily implemented in C# with the help of a little LINQ.

Adding the Add to Cart Buttons

We need to edit the Views/Shared/ProductSummary.cshtml partial view to add the buttons to the product listings. To prepare for this, we first need to add a class file called *UrlExtensions.cs* to the Infrastructure folder and defines the extension method shown in Listing 77.

Listing 48. The Contents of the *UrlExtensions.cs* File in the Infrastructure Folder

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

using Microsoft.AspNetCore.Http;

namespace SportsStore.Infrastructure
{
    public static class UrlExtensions
    {
        public static string PathAndQuery(this HttpRequest request) =>
            request.QueryString.HasValue
                ? $"{request.Path}{request.QueryString}"
                : request.Path.ToString();
    }
}

```

The *PathAndQuery* extension method operates on the *HttpRequest* class, which ASP.NET uses to describe an HTTP request. The extension method generates a URL that the browser will be returned to after the cart has been updated, taking into account the query string if there is one. In Listing 49, we add the namespace that contains the extension method to the view imports file so that we can use it in the partial view.

Listing 49. Adding a Namespace in the *_ViewImports.cshtml* File

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

In Listing 50, we update the partial view that describes each product to contain an Add To Cart button.

Listing 50. Adding the Buttons to the *ProductSummary.cshtml* File View

```
@model Product
<div class="card m-3 rounded">
  <div class="card-block m-2">
    <div class="card-body">
      <h3 class="card-title">
        <strong>@Model.Name</strong>
      </h3>

      <p class="card-text lead">
        @Model.Description
      </p>

      <span class="float-left bg-success text-white rounded m-2 p-1">
        @Model.Price.ToString("c")
      </span>

      <form id="@Model.ProductID" asp-action="AddToCart"
        asp-controller="Cart" method="post" class="float-right m-2 p-1">
        <input type="hidden" asp-for="ProductID" />
        <input type="hidden" name="returnUrl"
          value="@ViewContext.HttpContext.Request.PathAndQuery()" />
        <button type="submit" class="btn btn-info btn-sm float-right">
          Add To Cart
        </button>
      </form>
    </div>
  </div>
</div>
```

We have added a *form* element that contains hidden *input* elements specifying the *ProductID* value from the view model and the URL that the browser should be returned to after the cart has been updated. The *form* element and one of the *input* elements are configured using builtin tag helpers, which are a useful way of generating forms that contain model values and that target controllers and actions in the application. The other *input* element uses the extension method we created to set the return URL. We also added a *button* element that will submit the form to the application.

Note: Notice that we have set the method attribute on the *form* element to post, which instructs the browser to submit the form data using an HTTP POST request. You can change this so that forms use the GET method, but you should think carefully about doing so. The HTTP specification requires that GET requests must be idempotent, meaning that they must not cause changes, and adding a product to a cart is definitely a change.

1.41 Task 38: Enabling Sessions

We are going to store details of a user's cart using session state, which is data that is stored at the server and associated with a series of requests made by a user. ASP.NET provides a range of different ways to store session state, including storing it in memory, which is the approach that we are going to use. This has the advantage of simplicity, but it means that the session data is lost when the application is stopped or restarted.

The first step is to add new NuGet package to the SportsStore application. Listing 51 shows the addition we need to make to the SportsStore.csproj file.

Listing 51. Adding a Package to the SportsStore.csproj File in the SportsStore Project

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

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

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />
    <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="2.0.1" />

    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="2.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="2.0.1" />
    PrivateAssets="All" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.0.1" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="2.0.1" />

    <PackageReference Include="Microsoft.Extensions.Caching.Memory" Version="2.0.1" />

  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
      Version="2.0.1" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.1" />
  </ItemGroup>

</Project>
```

Enabling sessions requires adding services and middleware in the Startup class, as shown in Listing 52.

Listing 52. Enabling Sessions in the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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();
        }

        // This method gets called by the runtime. Use this method to add services to the
        // container.
        // For more information on how to configure your application, visit
        // https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
            services.AddMemoryCache();
            services.AddSession();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        // pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseSession();
            app.UseMvc(routes => {

                routes.MapRoute(
                    name: null,
                    template: "{category}/Page{page:int}",
                    defaults: new { controller = "Product", action = "List" }
                );
                routes.MapRoute(
                    name: null,
                    template: "Page{page:int}",
                    defaults: new { controller = "Product", action = "List", page = 1 }
                );
            });
        }
    }
}
```

```

    );
    routes.MapRoute(
        name: null,
        template: "{category}",
        defaults: new { controller = "Product", action = "List", page = 1 }
    );
    routes.MapRoute(
        name: null,
        template: "",
        defaults: new { controller = "Product", action = "List", page = 1 });

    routes.MapRoute(name: null, template: "{controller}/{action}/{id?}");
});
}
}
}

```

The AddMemoryCache method call sets up the in-memory data store. The AddSession method registers the services used to access session data, and the UseSession method allows the session system to automatically associate requests with sessions when they arrive from the client.

1.42 Task 39: Implementing the Cart Controller

We need a controller to handle the Add to Cart button presses. Add a new class file called CartController.cs to the Controllers folder and use it to define the class shown in Listing 82.

Listing 53. The Contents of the CartController.cs File in the Controllers Folder

```

using System.Linq;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Infrastructure;
using SportsStore.Models;

namespace SportsStore.Controllers
{
    public class CartController : Controller
    {
        private IProductRepository repository;

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

        public RedirectToActionResult AddToCart(int productId, string returnUrl)
        {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            if (product != null)
            {
                Cart cart = GetCart();
                cart.AddItem(product, 1);
                SaveCart(cart);
            }
            return RedirectToAction("Index", new { returnUrl });
        }
    }
}

```



```

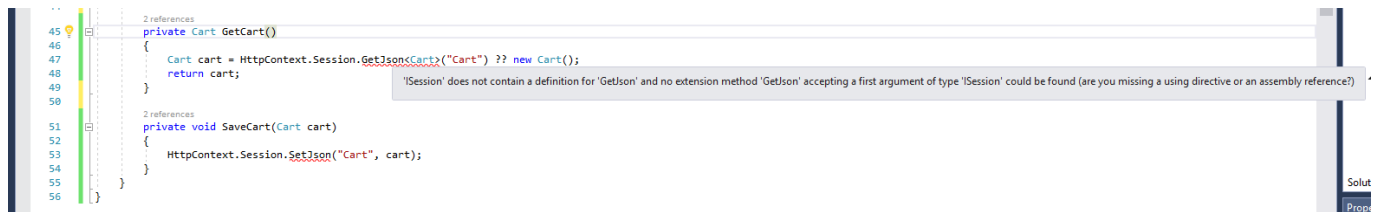
public RedirectToActionResult RemoveFromCart(int productId, string returnUrl)
{
    Product product = repository.Products
        .FirstOrDefault(p => p.ProductID == productId);

    if (product != null)
    {
        Cart cart = GetCart();
        cart.RemoveLine(product);
        SaveCart(cart);
    }
    return RedirectToAction("Index", new { returnUrl });
}

private Cart GetCart()
{
    Cart cart = HttpContext.Session.GetJson<Cart>("Cart") ?? new Cart();
    return cart;
}

private void SaveCart(Cart cart)
{
    HttpContext.Session.SetJson("Cart", cart);
}
}

```



'ISession' does not contain a definition for 'GetJson' and no extension method 'GetJson' accepting a first argument of type 'ISession' could be found (are you missing a using directive or an assembly reference)?

Figure 54. GetJson and SetJson Session methods errors

There are a few points to note about this controller. The first is that you will note Visual Studio shows errors for GetJson and SetJson methods as shown in Figure 54. These errors are occurring due to GetJson and SetJson Session Extensions are not defined, which we will next. The second is that we use the ASP.NET session state feature to store and retrieve Cart objects, which is the purpose of the GetCart method. The middleware that we registered in the previous section uses cookies or URL rewriting to associate multiple requests from a user together to form a single browsing session. A related feature is session state, which associates data with a session. This is an ideal fit for the Cart class: we want each user to have their own cart, and we want the cart to be persistent between requests. Data associated with a session is deleted when a session expires (typically because a user has not made a request for a while), which means that we do not need to manage the storage or life cycle of the Cart objects.

For the AddToCart and RemoveFromCart action methods, we have used parameter names that match the input elements in the HTML forms created in the ProductSummary.cshtml view. This allows MVC to

associate incoming form POST variables with those parameters, meaning we do not need to process the form. This is known as model binding and is a powerful tool for simplifying controller classes.

Defining Session State Extension Methods

The session state feature in ASP.NET Core stores only int, string, and byte[] values. Since we want to store a Cart object, we need to define extension methods to the ISession interface, which provides access to the session state data to serialize Cart objects into JSON and convert them back. We add a class file called SessionExtensions.cs to the Infrastructure folder and define the extension methods shown in Listing 54.

Listing 54. The Contents of the SessionExtensions.cs File in the Infrastructure Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Features;
using Newtonsoft.Json;
namespace SportsStore.Infrastructure
{
    public static class SessionExtensions
    {
        public static void SetJson(this ISession session, string key, object value)
        {
            session.SetString(key, JsonConvert.SerializeObject(value));
        }

        public static T GetJson<T>(this ISession session, string key)
        {
            var sessionData = session.GetString(key);
            return sessionData == null
                ? default(T) : JsonConvert.DeserializeObject<T>(sessionData);
        }
    }
}
```

These methods rely on the Json.Net package to serialize objects into the JavaScript Object Notation format. The Json.Net package doesn't have to be added to the package.json file because it is already used behind the scenes by MVC to provide the JSON helper feature (See www.newtonsoft.com/json for information on working directly with Json.Net).

The extension methods make it easy to store and retrieve Cart objects. To add a Cart to the session state in the controller, we make an assignment like this:

```
...
HttpContext.Session.SetJson("Cart", cart);
...
```

The HttpContext property is provided the Controller base class from which controllers are usually derived and returns an HttpContext object that provides context data about the request that has been received and the response that is being prepared. The HttpContext.Session property returns an object that implements the ISession interface, which is the type on which we defined the SetJson method, which accepts arguments that specify a key and an object that will be added to the session state. The extension method serializes the object and adds it to the session state using the underlying functionality provided by the ISession interface.

To retrieve the *Cart* again, we use the other extension method, specifying the same key, like this:

```
...
Cart cart = HttpContext.Session.GetJson<Cart>("Cart");
...
```

The type parameter lets us specify the type that we are expecting to be retrieved, which is used in the deserialization process.

1.43 Task 40: Displaying the Contents of the Cart

The final point to note about the *Cart* controller is that both the *AddToCart* and *RemoveFromCart* methods call the *RedirectToAction* method. This has the effect of sending an HTTP redirect instruction to the client browser, asking the browser to request a new URL. In this case, we have asked the browser to request a URL that will call the *Index* action method of the *Cart* controller.

We are going to implement the *Index* method and use it to display the contents of the *Cart*. If you refer back to Figure 53, you will see that this is the workflow when the user clicks the *Add to Cart* button.

We need to pass two pieces of information to the view that will display the contents of the cart: the *Cart* object and the URL to display if the user clicks the *Continue Shopping* button. We create a new class file called *CartIndexViewModel.cs* in the *Models/ViewModels* folder of the *SportsStore* project and used it to define the class shown in Listing 55.

Listing 55. The Contents of the *CartIndexViewModel.cs* File in the *Models/ViewModels* Folder

```
using SportsStore.Models;

namespace SportsStore.Models.ViewModels
{
    public class CartIndexViewModel
    {
        public Cart Cart { get; set; }

        public string returnUrl { get; set; }
    }
}
```

Now that we have the view model, we can implement the *Index* action method in the *Cart* controller class, as shown in Listing 56.

Listing 56. Implementing the *Index* Action Method in the *CartController.cs* File

```
using System.Linq;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Infrastructure;
using SportsStore.Models;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers
{
    public class CartController : Controller
```

```

{
    private IProductRepository repository;

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

    public ViewResult Index(string returnUrl)
    {
        return View(new CartIndexViewModel
        {
            Cart = GetCart(),
            ReturnUrl = returnUrl
        });
    }

    public RedirectToActionResult AddToCart(int productId, string returnUrl)
    {
        Product product = repository.Products
            .FirstOrDefault(p => p.ProductID == productId);
        if (product != null)
        {
            Cart cart = GetCart();
            cart.AddItem(product, 1);
            SaveCart(cart);
        }
        return RedirectToAction("Index", new { returnUrl });
    }

    public RedirectToActionResult RemoveFromCart(int productId, string returnUrl)
    {
        Product product = repository.Products
            .FirstOrDefault(p => p.ProductID == productId);

        if (product != null)
        {
            Cart cart = GetCart();
            cart.RemoveLine(product);
            SaveCart(cart);
        }
        return RedirectToAction("Index", new { returnUrl });
    }

    private Cart GetCart()
    {
        Cart cart = HttpContext.Session.GetJson<Cart>("Cart") ?? new Cart();
        return cart;
    }

    private void SaveCart(Cart cart)
    {
        HttpContext.Session.SetJson("Cart", cart);
    }
}

```

The Index action retrieves the Cart object from the session state and uses it to create a CartIndexViewModel object, which is then passed to the View method to be used as the view model.

The last step to display the contents of the cart is to create the view that the Index action will render. We create the Views/Cart folder and added to it a Razor view file called Index.cshtml with the markup shown in Listing 57.

Listing 57. The Contents of the Index.cshtml File in the Views/Cart Folder

```
@model CartIndexViewModel

<h2>Your cart</h2>

<table class="table table-bordered table-striped">
  <thead>
    <tr>
      <th class="text-center">Quantity</th>
      <th>Item</th>
      <th class="text-right">Price</th>
      <th class="text-right">Subtotal</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var line in Model.Cart.Lines)
    {
      <tr>
        <td class="text-center">@line.Quantity</td>
        <td class="text-left">@line.Product.Name</td>
        <td class="text-right">@line.Product.Price.ToString("c")</td>
        <td class="text-right">
          @((line.Quantity * line.Product.Price).ToString("c"))
        </td>
      </tr>
    }
  </tbody>
  <tfoot>
    <tr>
      <td colspan="3" class="text-right">Total:</td>
      <td class="text-right">
        @Model.Cart.ComputeTotalValue().ToString("c")
      </td>
    </tr>
  </tfoot>
</table>

<div class="text-center">
  <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
</div>
```

The view enumerates the lines in the cart and adds rows for each of them to an HTML table, along with the total cost per line and the total cost for the cart. The classes we have assigned the elements to correspond to Bootstrap styles for tables and text alignment.

The result is the basic functions of the shopping cart are in place. First, products are listed along with a button to add them to the cart, as shown in Figure 55.

And second, when the user clicks the Add to Cart button, the appropriate product is added to their cart, and a summary of the cart is displayed, as shown in Figure 56. Clicking the Continue Shopping button returns the user to the product page they came from.

In summary, we started to flesh out the customer-facing parts of the SportsStore app. We provided the means by which the user can navigate by category and put the basic building blocks in place for adding items to a shopping cart.

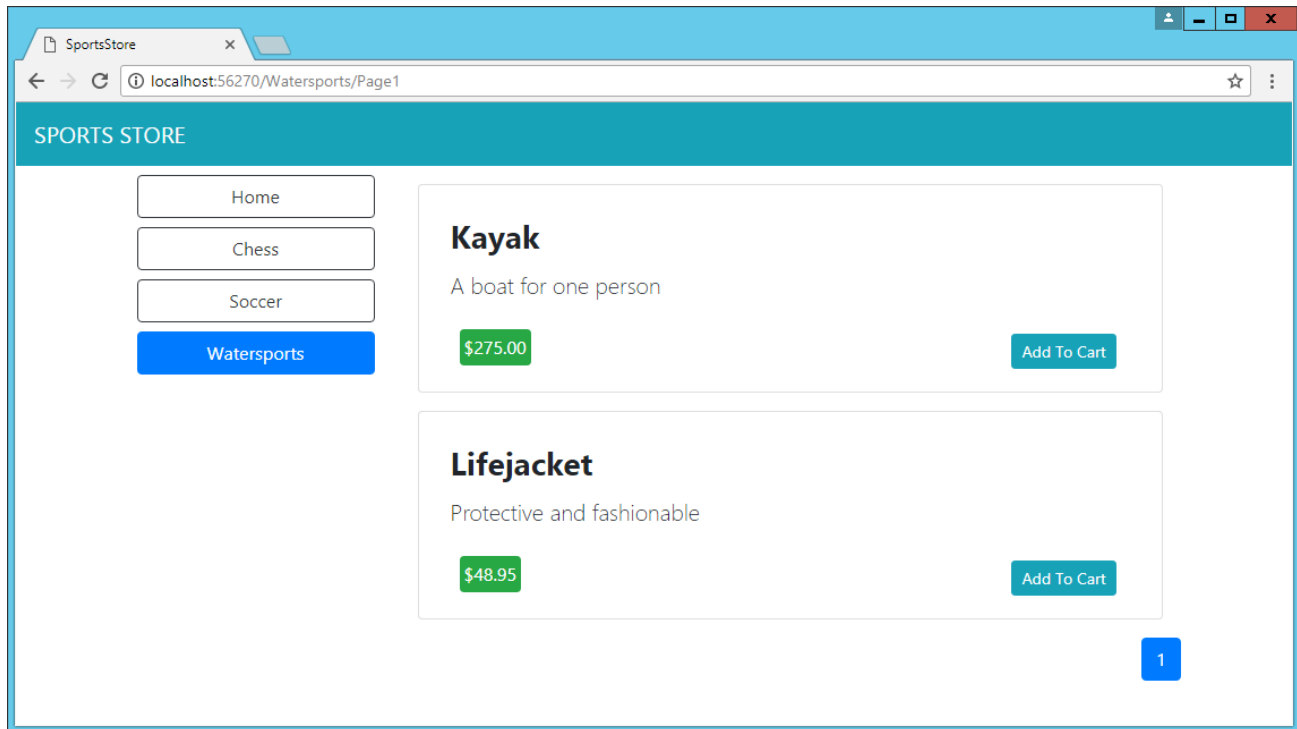


Figure 55. The Add to Cart button

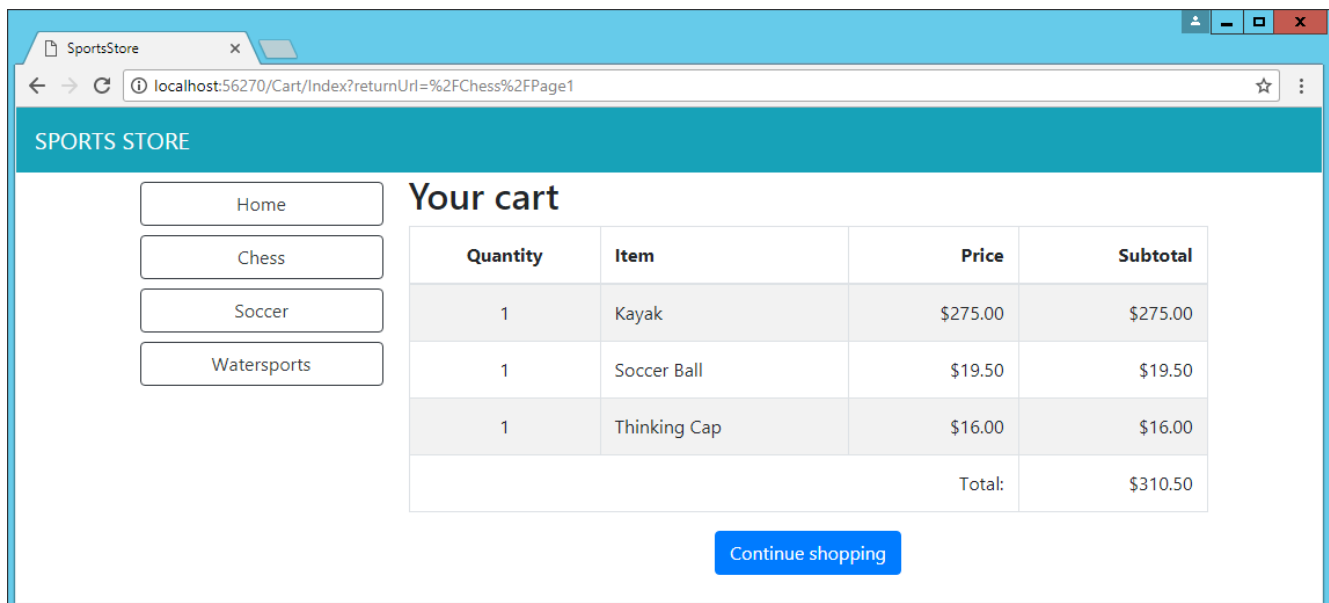


Figure 56. Displaying the contents of the shopping cart

1.44 Refining the Cart Model with a Service

We defined a Cart model class in the previous chapter and demonstrated how it can be stored using the session feature, allowing the user to build up a set of products for purchase. The responsibility for managing the persistence of the Cart class fell to the Cart controller, which explicitly defines methods for getting and storing Cart objects.

The problem with this approach is that we will have to duplicate the code that obtains and stores Cart objects in any component that uses them. In this section, we are going to use the services feature that sits at the heart of ASP.NET Core to simplify the way that Cart objects are managed, freeing individual components such as the Cart controller from needing to deal with the details directly.

Services are most commonly used to hide details of how interfaces are implemented from the components that depend on them. You have seen an example of this when we created a service for the `IProductRepository` interface, which allowed us to seamlessly replace the fake repository class with the Entity Framework Core repository. But services can be used to solve lots of other problems as well and can be used to shape and reshape an application, even when you are working with concrete classes such as Cart.

1.45 Task 41: Creating a Storage-Aware Cart Class

The first step in tidying up the way that the Cart class is used will be to create a subclass that is aware of how to store itself using session state. We create a class file called `SessionCart.cs` to the Models folder and use it to define the class shown in Listing 58.

Listing 58. The Contents of the SessionCart.cs File in the Models Folder

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Newtonsoft.Json;
using SportsStore.Infrastructure;

namespace SportsStore.Models
{
    public class SessionCart : Cart
    {
        public static Cart GetCart(IServiceProvider services)
        {
            ISession session = services.GetRequiredService<IHttpContextAccessor>().
                HttpContext.Session;
            SessionCart cart = session?.GetJson<SessionCart>("Cart")
                ?? new SessionCart();
            cart.Session = session;
            return cart;
        }

        [JsonIgnore]
        public ISession Session { get; set; }

        public override void AddItem(Product product, int quantity)
        {
            base.AddItem(product, quantity);
            Session.SetJson("Cart", this);
        }
    }
}
```

```

    }

    public override void RemoveLine(Product product)
    {
        base.RemoveLine(product);
        Session.SetJson("Cart", this);
    }

    public override void Clear()
    {
        base.Clear();
        Session.Remove("Cart");
    }
}
}

```

The SessionCart class subclasses the Cart class and overrides the AddItem, RemoveLine, and Clear methods so they call the base implementations and then store the updated state in the session using the extension methods on the ISession interface. The static GetCart method is a factory for creating SessionCart objects and providing them with an ISession object so they can store themselves.

Getting hold of the ISession object is a little complicated. We have to obtain an instance of the IHttpContextAccessor service, which provides me with access to an HttpContext object that, in turn, provides us with the ISession. This around-about approach is required because the session isn't provided as a regular service.

Registering the Service

The next step is to create a service for the Cart class. Goal is to satisfy requests for Cart objects with SessionCart objects that will seamlessly store themselves. You can see how we create the service in Listing 88 by editing Startup.cs file.

Listing 59. Creating the Cart Service in the Startup.cs File

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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();
    }

    // This method gets called by the runtime. Use this method to add services to the
    container.
    // For more information on how to configure your application, visit
    https://go.microsoft.com/fwlink/?LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
        services.AddTransient<IProductRepository, EFProductRepository>();
        services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
        services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
        services.AddMvc();
        services.AddMemoryCache();
        services.AddSession();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request
    pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory)
    {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
        app.UseStaticFiles();
        app.UseSession();
        app.UseMvc(routes => {

            routes.MapRoute(
                name: null,
                template: "{category}/Page{page:int}",
                defaults: new { controller = "Product", action = "List" }
            );
            routes.MapRoute(
                name: null,
                template: "Page{page:int}",
                defaults: new { controller = "Product", action = "List", page = 1 }
            );
            routes.MapRoute(
                name: null,
                template: "{category}",
                defaults: new { controller = "Product", action = "List", page = 1 }
            );
            routes.MapRoute(
                name: null,
                template: "",
                defaults: new { controller = "Product", action = "List", page = 1 });

            routes.MapRoute(name: null, template: "{controller}/{action}/{id?}");
        });
    }
}

```

The AddScoped method specifies that the same object should be used to satisfy related requests for Cart instances. How requests are related can be configured, but by default it means that any Cart required by components handling the same HTTP request will receive the same object.

Rather than provide the `AddScoped` method with a type mapping, as we did for the repository, we have specified a lambda expression that will be invoked to satisfy `Cart` requests. The expression receives the collection of services that have been registered and passes the collection to the `GetCart` method of the `SessionCart` class. The result is that requests for the `Cart` service will be handled by creating `SessionCart` objects, which will serialize themselves as session data when they are modified.

We also added a service using the `AddSingleton` method, which specifies that the same object should always be used. The service we created tells MVC to use the `HttpContextAccessor` class when implementations of the `IHttpContextAccessor` interface are required. This service is required so we can access the current session in the `SessionCart` class in Listing 58.

Simplifying the Cart Controller

The benefit of creating this kind of service is that it allows us to simplify the controllers where `Cart` objects are used. In Listing 60, we have reworked the `CartController` class to take advantage of the new service.

Listing 60. Using the Cart Service in the CartController.cs File

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

namespace SportsStore.Controllers
{
    public class CartController : Controller
    {
        private IProductRepository repository;

        private Cart cart;

        public CartController(IProductRepository repo, Cart cartService)
        {
            repository = repo;
            cart = cartService;
        }

        public ViewResult Index(string returnUrl)
        {
            return View(new CartIndexViewModel
            {
                Cart = cart,
                ReturnUrl = returnUrl
            });
        }

        public RedirectToActionResult AddToCart(int productId, string returnUrl)
        {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);

            if (product != null)
            {
                cart.AddItem(product, 1);
            }
            return RedirectToAction("Index", new { returnUrl });
        }
    }
}
```

```

public RedirectToActionResult RemoveFromCart(int productId,
string returnUrl)
{
    Product product = repository.Products
        .FirstOrDefault(p => p.ProductID == productId);

    if (product != null)
    {
        cart.RemoveLine(product);
    }
    return RedirectToAction("Index", new { returnUrl });
}
}

```

The CartController class indicates that it needs a Cart object by declaring a constructor argument, which has allowed us to remove the methods that read and write data from the session and the steps required to write updates. The result is a controller that is simpler and remains focused on its role in the application without having to worry about how Cart objects are created or persisted. And, since services are available throughout the application, any component can get hold of the user's cart using the same technique.

Completing the Cart Functionality

Now that we have introduced the Cart service, it is time to complete the cart functionality by adding two new features. The first will allow the customer to remove an item from the cart. The second feature will display a summary of the cart at the top of the page.

1.46 Task 42: Removing Items from the Cart

We already defined and tested the RemoveFromCart action method in the controller, so letting the customer remove items is just a matter of exposing this method in a view, which we are going to do by adding a Remove button in each row of the cart summary. Listing 90 shows the changes to Views/Cart/Index.cshtml.

Listing 61. Introducing a Remove Button to the Index.cshtml File in the Views/Cart Folder

```

@model CartIndexViewModel

<h2>Your cart</h2>

<table class="table table-bordered table-striped">
    <thead>
        <tr>
            <th>Quantity</th>
            <th>Item</th>
            <th class="text-right">Price</th>
            <th class="text-right">Subtotal</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var line in Model.Cart.Lines)
        {
            <tr>

```

```
 @line.Quantity</td>  @line.Product.Name</td>  @line.Product.Price.ToString("c")</td>  @((line.Quantity * line.Product.Price).ToString("c")) </td> Total:</td>  @Model.Cart.ComputeTotalValue().ToString("c") </td> </tr> </tfoot> </table> <div class="text-center"> <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a> </div> | | | | |
```

We added a new column to each row of the table that contains a form with hidden input elements that specify the product to be removed and the return URL, along with a button that submits the form.

You can see the Remove buttons at work by running the application and adding items to the shopping cart. Remember that the cart already contains the functionality to remove it, which you can test by clicking one of the new buttons, as shown in Figure 57.

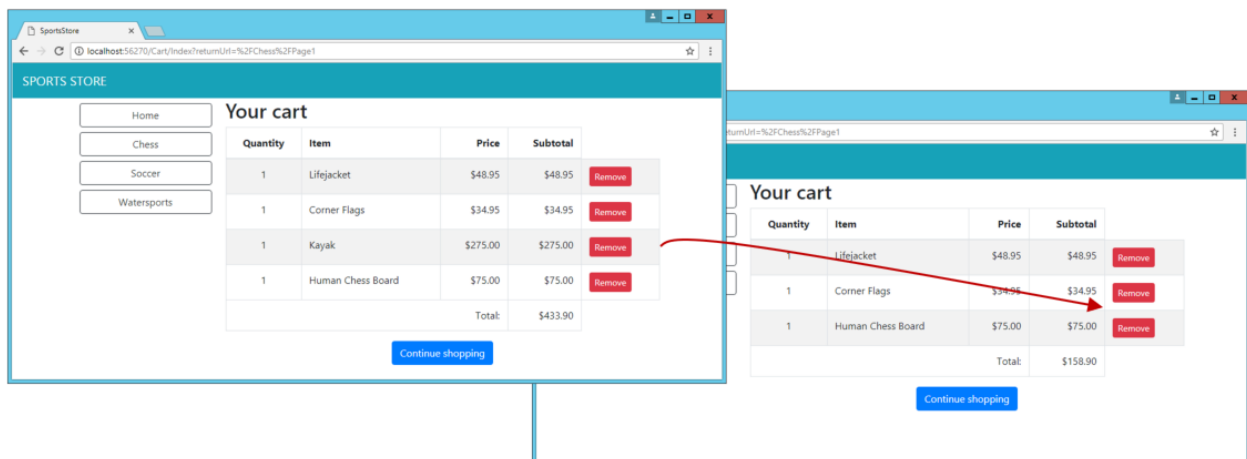


Figure 57. Removing an item from the shopping cart

1.47 Task 43: Adding the Cart Summary Widget

We may have a functioning cart, but there is an issue with the way it is integrated into the interface. Customers can tell what is in their cart only by viewing the cart summary screen. And they can view the cart summary screen only by adding a new item to the cart.

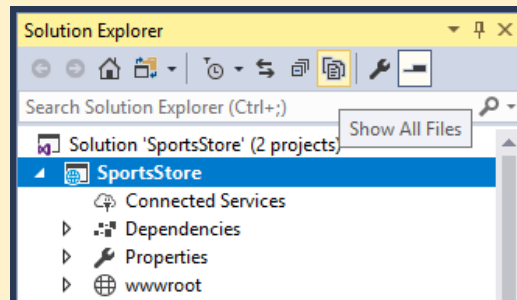
To solve this problem, we are going to add a widget that summarizes the contents of the cart and that can be clicked to display the cart contents throughout the application. We will do this in much the same way that we added the navigation widget—as a view component whose output we can include in the Razor shared layout.

Adding the Font Awesome Package

As part of the cart summary, we are going to display a button that allows the user to check out. Rather than display the word checkout in the button, we want to use a cart symbol. We are going to use the Font Awesome package, which is an excellent set of open source icons that are integrated into applications as fonts, where each character in the font is a different image. You can learn more about Font Awesome, including inspecting the icons it contains, at <http://fontawesome.github.io/Font-Awesome>.

We add the Font Awesome package to the dependencies section in the bower.json file, as shown in Listing 62. When the bower.json file is saved, Visual Studio uses Bower to download and install the Font Awesome package in the `www/lib/fontawesome` folder.

Note: In case bower.json file is hidden, then select the SportsStore project and click the Show All Items button at the top of the Solution Explorer to reveal the bower.json file as shown in the below figure.



Listing 62. Adding the Font Awesome Package in the bower.json File

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "jquery": "3.3.1",
    "bootstrap": "4.0.0",
    "fontawesome": "5.0.8"
  }
}
```

Creating the View Component Class and View

We add a class file called `CartSummaryViewComponent.cs` in the Components folder and use it to define the view component shown in Listing 63.

Listing 63. The Contents of the CartSummaryViewComponent.cs File in the Components Folder

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

namespace SportsStore.Components
{
    public class CartSummaryViewComponent : ViewComponent
    {
        private Cart cart;

        public CartSummaryViewComponent(Cart cartService)
        {
            cart = cartService;
        }

        public IViewComponentResult Invoke()
        {
            return View(cart);
        }
    }
}
```

This view component is able to take advantage of the service that we created earlier in order to receive a Cart object as a constructor argument. The result is a simple view component class that passes on the Cart to the View method in order to generate the fragment of HTML that will be included in the layout. To create the layout, we create the Views/Shared/Components/CartSummary folder, added to it a Razor view file called Default.cshtml, and added the markup shown in Listing 93.

Listing 64. The Default.cshtml File in the Views/Shared/Components/CartSummary Folder

```
@model Cart

<div class="">
    @if (Model.Lines.Count() > 0)
    {
        <small class="navbar-text">
            <b>Your cart:</b>
            @Model.Lines.Sum(x => x.Quantity) item(s)
            @Model.ComputeTotalValue().ToString("c")
        </small>
    }
    <a class="btn btn-sm btn-dark"
        asp-controller="Cart" asp-action="Index"
        asp-route-returnurl="@ViewContext.HttpContext.Request.PathAndQuery()">
        <i class="fa fa-shopping-cart"></i>
    </a>
</div>
```

The view displays a button with the Font Awesome cart icon and, if there are items in the cart, provides a snapshot that details the number of items and their total value. Now that we have a view component and a view, we can modify the shared layout so that the cart summary is included in the responses generated by the application's controllers, as shown in Listing 65.

Listing 65. Adding the Cart Summary in the _Layout.cshtml File

```
</html>
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <link rel="stylesheet" asp-href-include="lib/fontawesome/web-fonts-with-css/css/*.css" />
  <title>SportsStore</title>
</head>
<body>
  <nav class="navbar navbar-inverse bg-info justify-content-between">
    <a class="navbar-brand text-white" href="#">SPORTS STORE</a>
    <div class="float-right">
      @await Component.InvokeAsync("CartSummary")
    </div>
  </nav>
  <div class="container">
    <div class="row">
      <div class="col col-md-3">
        @await Component.InvokeAsync("NavigationMenu")
      </div>
      <div class="col col-md-9">
        @RenderBody()
      </div>
    </div>
  </div>
</body>
</html>
```

You can see the cart summary by starting the application. When the cart is empty, only the checkout button is shown. If you add items to the cart, then the number of items and their combined cost are shown, as illustrated by Figure 58. With this addition, customers know what is in their cart and have an obvious way to check out from the store.

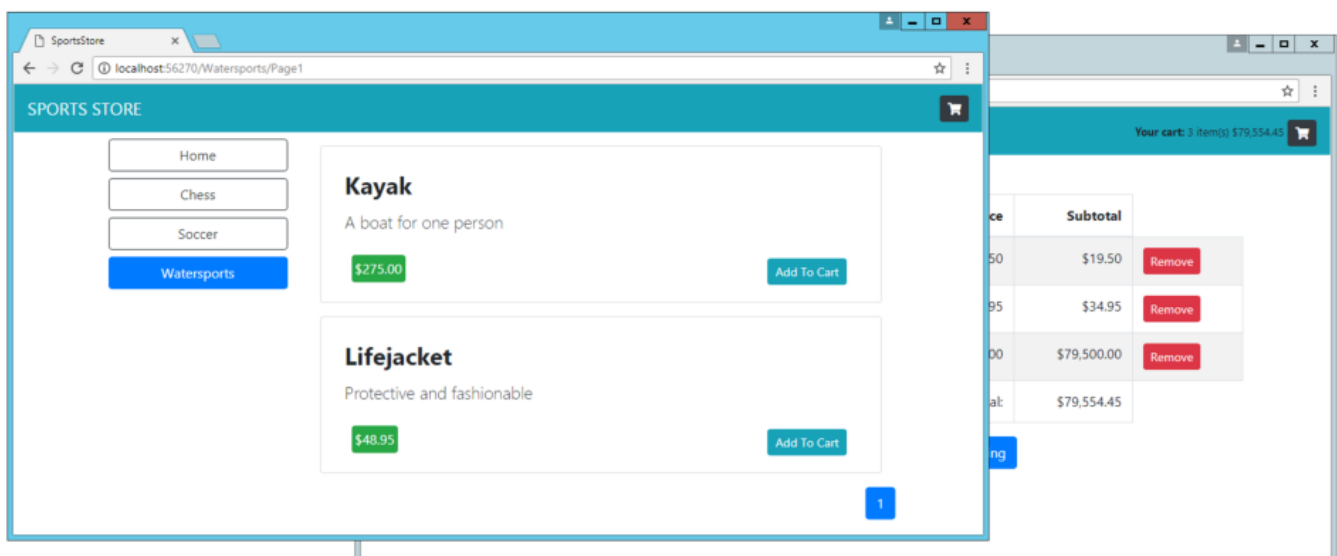


Figure 58. Displaying a summary of the cart

1.48 Submitting Orders

We have now reached the final customer feature in SportsStore: the ability to check out and complete an order. In the following sections, we will extend the domain model to provide support for capturing the shipping details from a user and add the application support to process those details.

1.49 Task 44: Creating the Model Class

We add a class file called Order.cs to the Models folder and edited it to match the contents shown in Listing 66. This is the class we will use to represent the shipping details for a customer.

Listing 66. The Contents of the Order.cs File in the Models Folder

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models
{
    public class Order
    {
        [BindNever]
        public int OrderID { get; set; }

        [BindNever]
        public ICollection<CartLine> Lines { get; set; }

        [Required(ErrorMessage = "Please enter a name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter the first address line")]
        public string Line1 { get; set; }

        public string Line2 { get; set; }

        public string Line3 { get; set; }

        [Required(ErrorMessage = "Please enter a city name")]
        public string City { get; set; }

        [Required(ErrorMessage = "Please enter a state name")]
        public string State { get; set; }

        public string Zip { get; set; }

        [Required(ErrorMessage = "Please enter a country name")]
        public string Country { get; set; }

        public bool GiftWrap { get; set; }
    }
}
```

We are using the validation attributes from the System.ComponentModel.DataAnnotations namespace. See below note on Model Validation for more details. We also use the BindNever attribute, which prevents the user supplying values for these properties in an HTTP request. This is a feature of the model binding system.

Note: Understanding Model Validation

Model validation is the process of ensuring the data received by the application is suitable for binding to the model and, when this is not the case, providing useful information to the user that will help explain the problem.

The first part of the process, checking the data received, is one of the key ways to preserve the integrity of the domain model. Rejecting data that doesn't make sense in the context of the domain can prevent odd and unwanted states from arising in the application. The second part, helping the user correct the problem, is equally important. Without the information and feedback they need to interact with the application, users become frustrated and confused. In public-facing applications, this means users will simply stop using the application. In corporate applications, this means the user's workflow will be hindered. Neither outcome is desirable, but fortunately MVC provides extensive support for model validation. Below table puts model validation in context.

Question	Answer
What are they?	Model validation is the process of ensuring that the data provided in a request is valid for use in the application.
Why are they useful?	Users do not always enter valid data, and using it in the application can produce unexpected and undesirable errors.
How are they used?	Controllers check the outcome of the validation process, and tag helpers are used to include validation feedback in views displayed to the user. Validation is performed automatically during the model binding process and is usually supplemented with custom validation in a controller class or by using validation attributes.
Are there any pitfalls or limitations?	It is important to test the efficacy of your validation code to ensure that it prevents against the full range of values that the application can receive.
Are there any alternatives?	No, model validation is tightly integrated into ASP.NET Core MVC.
Have they changed since MVC 5?	The basic approach to performing validation remains the same as in earlier versions of MVC, but some of the underlying class and interfaces have changed.

Below table shows the set of built-in validation attributes available in an MVC application.

Attribute	Example	Description
Compare	[Compare("OtherProperty")]	This attribute ensures that properties must have the same value, which is useful when you ask the user to provide the same information twice, such as an e-mail address or a password.
Range	[Range(10, 20)]	This attribute ensures that a numeric value (or any property type that implements <code>IComparable</code>) does not lie beyond the specified minimum and maximum values. To specify a boundary on only one side, use a <code>MinValue</code> or <code>MaxValue</code> constant—for example, <code>[Range(int.MinValue, 50)]</code> .

RegularExpression	[RegularExpression("pattern")]	This attribute ensures that a string value matches the specified regular expression pattern. Note that the pattern has to match the entire user-supplied value, not just a substring within it. By default, it matches case sensitively, but you can make it case insensitive by applying the (?i) modifier—that is, [RegularExpression("(?i)mypattern")].
Required	[Required]	This attribute ensures that the value is not empty or a string consisting only of spaces. If you want to treat whitespace as valid, use [Required(AllowEmptyStrings = true)] .
StringLength	[StringLength(10)]	This attribute ensures that a string value is not longer than a specified maximum length. You can also specify a minimum length: [StringLength(10, MinimumLength=2)] .

1.50 Task 45: Adding the Checkout Process

The goal is to reach the point where users are able to enter their shipping details and submit their order. To start, we need to add a Checkout button to the cart summary view. Listing 67 shows the change we need to apply to the Views/Cart/Index.cshtml file.

Listing 67. Adding the Checkout Now Button to the Index.cshtml File in the Views/Cart Folder

@model CartIndexViewModel

<h2>Your cart</h2>

```
<table class="table table-bordered table-striped">
  <thead>
    <tr>
      <th class="text-center">Quantity</th>
      <th>Item</th>
      <th class="text-right">Price</th>
      <th class="text-right">Subtotal</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var line in Model.Cart.Lines)
    {
      <tr>
        <td class="text-center">@line.Quantity</td>
        <td class="text-left">@line.Product.Name</td>
        <td class="text-right">@line.Product.Price.ToString("c")</td>
        <td class="text-right">
          @((line.Quantity * line.Product.Price).ToString("c"))
        </td>
        <td>
          <form asp-action="RemoveFromCart" method="post">
            <input type="hidden" name="ProductID"
              value="@line.Product.ProductID" />
            <input type="hidden" name="returnUrl"

```

```

        value="@Model.ReturnUrl" />
        <button type="submit" class="btn btn-sm btn-danger ">
            Remove
        </button>
    </form>
</td>

</tr>
}
</tbody>
<tfoot>
    <tr>
        <td colspan="3" class="text-right">Total:</td>
        <td class="text-right">
            @Model.Cart.ComputeTotalValue().ToString("c")
        </td>
    </tr>
</tfoot>
</table>

<div class="text-center">
    <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
    <a class="btn btn-primary" asp-action="Checkout" asp-controller="Order">
        Checkout
    </a>
</div>

```

This change generates a link that we have styled as a button and that, when clicked, calls the Checkout action method of the Order controller, which we create in the following section. You can see how this button appears in Figure 59.

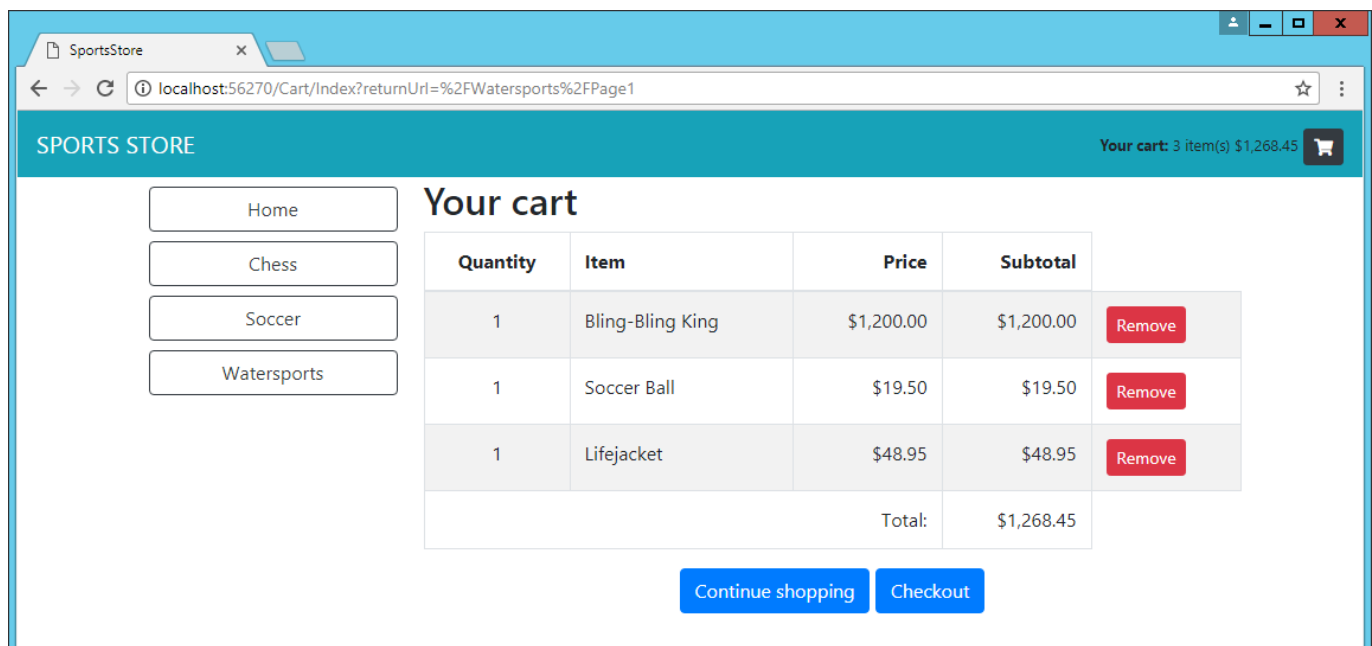


Figure 59. The Checkout button

We now need to define the Order controller. We add a class file called `OrderController.cs` to the Controllers folder and used it to define the class shown in Listing 68.

Listing 68. The Contents of the OrderController.cs File in the Controllers Folder

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

namespace SportsStore.Controllers
{
    public class OrderController : Controller
    {
        public ViewResult Checkout() => View(new Order());
    }
}
```

The Checkout method returns the default view and passes a new ShippingDetails object as the view model. To create the view, we create the Views/Order folder and added a Razor view file called Checkout.cshtml with the markup shown in Listing 98.

Listing 69. The Contents of the Checkout.cshtml File in the Views/Order Folder

```
@model Order

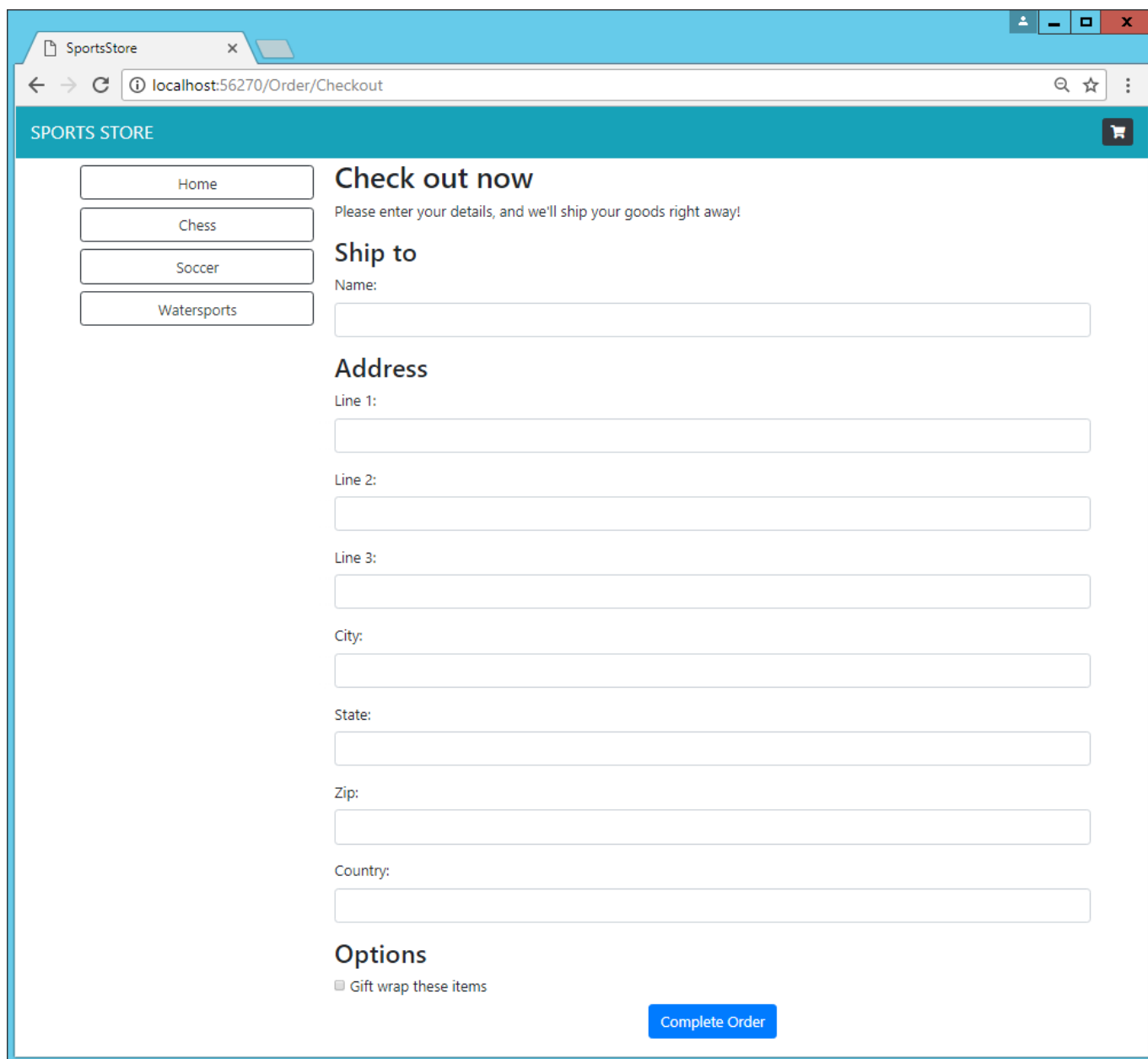
<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>

<form asp-action="Checkout" method="post">
    <h3>Ship to</h3>
    <div class="form-group">
        <label>Name:</label><input asp-for="Name" class="form-control" />
    </div>
    <h3>Address</h3>
    <div class="form-group">
        <label>Line 1:</label><input asp-for="Line1" class="form-control" />
    </div>
    <div class="form-group">
        <label>Line 2:</label><input asp-for="Line2" class="form-control" />
    </div>
    <div class="form-group">
        <label>Line 3:</label><input asp-for="Line3" class="form-control" />
    </div>
    <div class="form-group">
        <label>City:</label><input asp-for="City" class="form-control" />
    </div>
    <div class="form-group">
        <label>State:</label><input asp-for="State" class="form-control" />
    </div>
    <div class="form-group">
        <label>Zip:</label><input asp-for="Zip" class="form-control" />
    </div>
    <div class="form-group">
        <label>Country:</label><input asp-for="Country" class="form-control" />
    </div>
    <h3>Options</h3>
    <div class="checkbox">
        <label>
            <input asp-for="GiftWrap" /> Gift wrap these items
        </label>
    </div>
    <div class="text-center">
        <input class="btn btn-primary" type="submit" value="Complete Order" />
    </div>
</form>
```

</form>

For each of the properties in the model, we create a label and input element to capture the user input, formatted with Bootstrap. The asp-for attribute on the input elements is handled by a built-in tag helper that generates the type, id, name, and value attributes based on the specified model property.

You can see the effect of the new action method and view by starting the application, clicking the cart button at the top of the page, and then clicking the Checkout button, as shown in Figure 60. You can also reach this point by requesting the /Order/Checkout URL.



The screenshot shows a web browser window with the address bar set to `localhost:56270/Order/Checkout`. The page has a teal header with "SPORTS STORE" and a shopping cart icon. On the left, there's a sidebar with buttons for "Home", "Chess", "Soccer", and "Watersports". The main content area is titled "Check out now" with a sub-header "Ship to". Below this, there are several input fields for shipping information: "Name:", "Address" (with "Line 1:", "Line 2:", and "Line 3:" labels), "City:", "State:", "Zip:", and "Country:". There is also a checkbox labeled "Gift wrap these items" under the "Options" section. A blue "Complete Order" button is at the bottom right.

Figure 60. The shipping details form

1.51 Task 46: Implementing Order Processing

We will process orders by writing them to the database. Most e-commerce sites would not simply stop there, of course, and we have not provided support for processing credit cards or other forms of payment. But we want to keep things focused on MVC, so a simple database entry will do.

Extending the Database

Adding a new kind of model to the database is simple once the basic plumbing for entity framework is already created. First, we add a new property to the database context class in the Models folder, as shown in Listing 70.

Listing 70. Adding a Property in the ApplicationDbContext.cs File in Models folder

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

using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) :
base(options) { }
        public DbSet<Product> Products { get; set; }
        public DbSet<Order> Orders { get; set; }
    }
}
```

This change is enough of a foundation for Entity Framework Core to create a database migration that will allow Order objects to be stored in the database. To create the migration, open the Package Manger Console from the Tools > NuGet Package Manage menu and run the following command:

Add-Migration Orders

This command tells EF Core to take a new snapshot of the application, work out how it differs from the previous database version, and generate a new migration called Orders. To update the database schema, run the following command:

Update-Database

Creating the Order Repository

We are going to follow the same pattern we used for the product repository to provide access to the Order objects. We add an interface file called IOrderRepository.cs to the Models folder and use it to define the interface shown in Listing 71.

Listing 71. The Contents of the IOrderRepository.cs File in the Models Folder

```
using System.Collections.Generic;

namespace SportsStore.Models
{
    public interface IOrderRepository
    {
        IEnumerable<Order> Orders { get; }
        void SaveOrder(Order order);
    }
}
```

To implement the order repository interface, we add a class file called EOrderRepository.cs to the Models folder and defined the class shown in Listing 72. This class implements the IOrderRepository using Entity Framework Core, allowing the set of Order objects that have been stored to be retrieved and for orders to be created or changed.

Listing 72. The Contents of the EOrderRepository.cs File in the Models Folder

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

namespace SportsStore.Models
{
    public class EOrderRepository : IOrderRepository
    {
        private ApplicationDbContext context;
        public EOrderRepository(ApplicationDbContext ctx)
        {
            context = ctx;
        }
        public IEnumerable<Order> Orders => context.Orders
            .Include(o => o.Lines)
            .ThenInclude(l => l.Product);
        public void SaveOrder(Order order)
        {
            context.AttachRange(order.Lines.Select(l => l.Product));
            if (order.OrderID == 0)
            {
                context.Orders.Add(order);
            }
            context.SaveChanges();
        }
    }
}
```

In Listing 73, we register the order repository as a service in the ConfigureServices method of the Startup class.

Listing 73. Registering the Order Repository Service in the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```

```

using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
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();
        }

        // This method gets called by the runtime. Use this method to add services to the
        container.
        // For more information on how to configure your application, visit
        https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
            services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
            services.AddTransient<IOrderRepository, EFOrderRepository>();
            services.AddMvc();
            services.AddMemoryCache();
            services.AddSession();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseSession();
            app.UseMvc(routes => {

                routes.MapRoute(
                    name: null,
                    template: "{category}/Page{page:int}",
                    defaults: new { controller = "Product", action = "List" }
                );
                routes.MapRoute(
                    name: null,
                    template: "Page{page:int}",
                    defaults: new { controller = "Product", action = "List", page = 1 }
                );
                routes.MapRoute(
                    name: null,
                    template: "{category}",
                    defaults: new { controller = "Product", action = "List", page = 1 }
                );
            });
        }
    }
}

```



```

    );
    routes.MapRoute(
        name: null,
        template: "",
        defaults: new { controller = "Product", action = "List", page = 1 });

    routes.MapRoute(name: null, template: "{controller}/{action}/{id?}");

});

}
}
}

```

1.52 Task 47: Completing the Order Controller

To complete the `OrderController` class, we need to modify the constructor so that it receives the services it requires to process an order and add a new action method that will handle the HTTP form POST request when the user clicks the Complete Order button. Listing 74 shows both changes.

Listing 74. Completing the Controller in the `OrderController.cs` File

```

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

namespace SportsStore.Controllers
{
    public class OrderController : Controller
    {
        private IOrderRepository repository;
        private Cart cart;
        public OrderController(IOrderRepository repoService, Cart cartService)
        {
            repository = repoService;
            cart = cartService;
        }

        public ViewResult Checkout() => View(new Order());

        [HttpPost]
        public IActionResult Checkout(Order order)
        {
            if (cart.Lines.Count() == 0)
            {
                ModelState.AddModelError("", "Sorry, your cart is empty!");
            }
            if (ModelState.IsValid)
            {
                order.Lines = cart.Lines.ToArray();
                repository.SaveOrder(order);
                return RedirectToAction(nameof(Completed));
            }
            else
            {
                return View(order);
            }
        }
    }
}

```

```

        public ViewResult Completed()
        {
            cart.Clear();
            return View();
        }
    }
}

```

The Checkout action method is decorated with the `HttpPost` attribute, which means that it will be invoked for a POST request—in this case, when the user submits the form. Once again, we rely on the model binder system so that we can receive the `Order` object, which we then complete using data from the `Cart` and store in the repository.

MVC checks the validation constraints that we applied to the `Order` class using the data annotation attributes, and any validation problems violations are passed to the action method through the `ModelState` property. We can see whether there are any problems by checking the `ModelState.IsValid` property. We call the `ModelState.AddModelError` method to register an error message if there are no items in the cart.

1.53 Task 48: Displaying Validation Errors

MVC will use the validation attributes applied to the `Order` class to validate user data. However, we need to make a simple change to display any problems. This relies on another builtin tag helper that inspects the validation state of the data provided by the user and adds warning messages for each problem that has been discovered. Listing 75 shows the addition of an HTML element that will be processed by the tag helper to the `Checkout.cshtml` file.

Listing 75. Adding a Validation Summary to the Checkout.cshtml File

`@model Order`

```

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

<div asp-validation-summary="All" class="text-danger"></div>

<form asp-action="Checkout" method="post">
    <h3>Ship to</h3>
    <div class="form-group">
        <label>Name:</label><input asp-for="Name" class="form-control" />
    </div>
    <h3>Address</h3>
    <div class="form-group">
        <label>Line 1:</label><input asp-for="Line1" class="form-control" />
    </div>
    <div class="form-group">
        <label>Line 2:</label><input asp-for="Line2" class="form-control" />
    </div>
    <div class="form-group">
        <label>Line 3:</label><input asp-for="Line3" class="form-control" />
    </div>
    <div class="form-group">
        <label>City:</label><input asp-for="City" class="form-control" />
    </div>
    <div class="form-group">
        <label>State:</label><input asp-for="State" class="form-control" />
    </div>

```

```

<div class="form-group">
  <label>Zip:</label><input asp-for="Zip" class="form-control" />
</div>
<div class="form-group">
  <label>Country:</label><input asp-for="Country" class="form-control" />
</div>
<h3>Options</h3>
<div class="checkbox">
  <label>
    <input asp-for="GiftWrap" /> Gift wrap these items
  </label>
</div>
<div class="text-center">
  <input class="btn btn-primary" type="submit" value="Complete Order" />
</div>
</form>

```

With this simple change, validation errors are reported to the user. To see the effect, go to the /Order/Checkout URL and try to check out without selecting any products or filling any shipping details, as shown in Figure 61. The tag helper that generates these messages is part of the model validation system.

The screenshot shows a web browser window with the URL `localhost:56270/Order/Checkout`. The page has a teal header with "SPORTS STORE" and a shopping cart icon. On the left, there are four buttons: "Home", "Chess", "Soccer", and "Watersports". The main content area is titled "Check out now" and includes the text "Please enter your details, and we'll ship your goods right away!". Below this, there is a list of validation messages in red:

- Please enter a name
- Please enter the first address line
- Please enter a city name
- Please enter a state name
- Please enter a country name
- Sorry, your cart is empty!

Below the messages, there are three sections for shipping information:

- Ship to**: A text input field for "Name:".
- Address**: Three text input fields for "Line 1:", "Line 2:", and "Line 3:".

Figure 61. Displaying validation messages

Note: Client-side Validation

The data submitted by the user is sent to the server before it is validated, which is known as server-side validation and for which MVC has excellent support. The problem with server-side validation is that the user isn't told about errors until after the data has been sent to the server and processed and the result page has been generated—something that can take a few seconds on a busy server. For this reason, server-side validation is usually complemented by client-side validation, where JavaScript is used to check the values that the user has entered before the form data is sent to the server.

MVC supports unobtrusive client-side validation. The term unobtrusive means that validation rules are expressed using attributes added to the HTML elements that views generate. These attributes are interpreted by a JavaScript library that is included as part of MVC that, in turn, configures the jQuery Validation library, which does the actual validation work. Using client-side validation means adding three JavaScript files to the view: the jQuery library, the jQuery validation library, and the Microsoft unobtrusive validation library.

1.54 Task 49: Displaying a Summary Page

To complete the checkout process, we need to create the view that will be shown when the browser is redirected to the Completed action on the Order controller. We add a Razor view file called Completed.cshtml to the Views/Order folder and add the markup shown in Listing 76.

Listing 76. The Contents of the Completed.cshtml File in the Views/Order Folder

```
<h2>Thanks!</h2>
<p>Thanks for placing your order.</p>
<p>We'll ship your goods as soon as possible.</p>
```

We don't need to make any code changes to integrate this view into the application because we already added the required statements when we defined the Completed action method in Listing 74 (OrderController.cs file). Now customers can go through the entire process, from selecting products to checking out. If they provide valid shipping details (and have items in their cart), they will see the summary page when they click the Complete Order button, as shown in Figure 62.

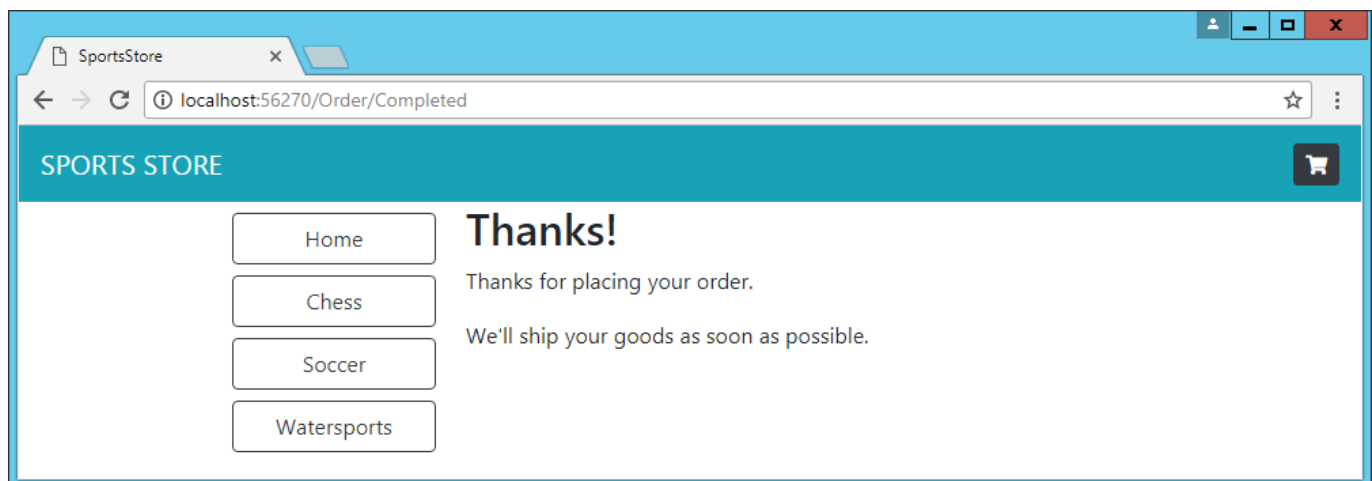


Figure 62. The completed order summary view

In summary, we have completed all the major parts of the customer-facing portion of SportsStore. It might not be enough to worry Amazon, but we have a product catalog that can be browsed by category and page, a neat shopping cart, and a simple checkout process.

The well-separated architecture means we can easily change the behavior of any piece of the application without worrying about causing problems or inconsistencies elsewhere. For example, we could change the way that orders are stored and it would not have any impact on the shopping cart, the product catalog, or any other area of the application.

1.55 Task 50: Managing Orders

In this section, we continue to build the SportsStore application in order to give the site administrator a way of managing orders and products. In the previous sessions, we added support for receiving orders from customers and storing them in a database. We now need to create a simple administration tool that will let us view the orders that have been received and mark them as shipped.

Enhancing the Model

The first change we need to make is to enhance the model so that we can record which orders have been shipped. Listing 77 shows the addition of a new property to the Order class, which is defined in the Order.cs file in the Models folder.

Listing 77. Adding a Property in the Order.cs File

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models
{
    public class Order
    {
        [BindNever]
        public int OrderID { get; set; }

        [BindNever]
        public ICollection<CartLine> Lines { get; set; }

        [BindNever]
        public bool Shipped { get; set; }

        [Required(ErrorMessage = "Please enter a name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter the first address line")]
        public string Line1 { get; set; }

        public string Line2 { get; set; }

        public string Line3 { get; set; }

        [Required(ErrorMessage = "Please enter a city name")]
        public string City { get; set; }

        [Required(ErrorMessage = "Please enter a state name")]
```

```

    public string State { get; set; }

    public string Zip { get; set; }

    [Required(ErrorMessage = "Please enter a country name")]
    public string Country { get; set; }

    public bool GiftWrap { get; set; }
}

```

This iterative approach of extending and adapting the model to support different features is typical of MVC development. In an ideal world, you would be able to completely define the model classes at the start of the project and just build the application around them, but that happens only for the simplest of projects and, in practice, iterative development is to be expected as the understanding of what is required develops and evolves.

Entity Framework Core migrations makes this process easier because you don't have to manually keep the database schema synchronized to the model class by writing your own SQL commands. To update the database to reflect the addition of the Shipped property to the Order class, open the Package Manager Console and run the following commands to create a new migration and apply it to the database:

```

Add-Migration ShippedOrders
Update-Database

```

1.56 Task 51: Adding the Actions and View

The functionality required to display and update the set of orders in the database is relatively simple because it builds on the features and infrastructure that we created in previous sessions. In Listing 78, we have added two new action methods to the Order controller.

Listing 78. Adding Action Methods in the OrderController.cs File

```

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

namespace SportsStore.Controllers
{
    public class OrderController : Controller
    {
        private IOrderRepository repository;
        private Cart cart;
        public OrderController(IOrderRepository repoService, Cart cartService)
        {
            repository = repoService;
            cart = cartService;
        }

        public ViewResult List() => View(repository.Orders.Where(o => !o.Shipped));

        [HttpPost]
        public IActionResult MarkShipped(int orderID)
        {
            Order order = repository.Orders

```

```

        .FirstOrDefault(o => o.OrderID == orderID);
        if (order != null)
        {
            order.Shipped = true;
            repository.SaveOrder(order);
        }
        return RedirectToAction(nameof(List));
    }

    public ViewResult Checkout() => View(new Order());

    [HttpPost]
    public IActionResult Checkout(Order order)
    {
        if (cart.Lines.Count() == 0)
        {
            ModelState.AddModelError("", "Sorry, your cart is empty!");
        }
        if (ModelState.IsValid)
        {
            order.Lines = cart.Lines.ToArray();
            repository.SaveOrder(order);
            return RedirectToAction(nameof(Completed));
        }
        else
        {
            return View(order);
        }
    }

    public ViewResult Completed()
    {
        cart.Clear();
        return View();
    }
}

```

The List method selects all the Order objects in the repository that have a Shipped value of false and passes them to the default view. This is the action method that we will use to display a list of the unshipped order to the administrator.

The MarkShipped method will receive a POST request that specifies the ID of an order, which is used to locate the corresponding Order object from the repository so that its Shipped property can be set to true and saved.

To display the list of unshipped orders, we add a Razor view file called List.cshtml to the Views/Order folder and added the markup shown in Listing 79. A table element is used to display some of the details from each other, including details of which products have been purchased.

Listing 79. The Contents of the List.cshtml File in the Views/Order Folder

```

@model IEnumerable<Order>

@{
    ViewBag.Title = "Orders";
    Layout = "_AdminLayout";
}

```

```

@if (Model.Count() > 0)
{
    <table class="table table-bordered table-striped">
        <tr>
            <th>Name</th>
            <th>Zip</th>
            <th colspan="2">Details</th>
            <th></th>
        </tr>
        @foreach (Order o in Model)
        {
            <tr>
                <td>@o.Name</td>
                <td>@o.Zip</td>
                <th>Product</th>
                <th>Quantity</th>
                <td>
                    <form asp-action="MarkShipped" method="post">
                        <input type="hidden" name="orderId" value="@o.OrderID" />
                        <button type="submit" class="btn btn-sm btn-danger">
                            Ship
                        </button>
                    </form>
                </td>
            </tr>
            @foreach (CartLine line in o.Lines)
            {
                <tr>
                    <td colspan="2"></td>
                    <td>@line.Product.Name</td>
                    <td>@line.Quantity</td>
                    <td></td>
                </tr>
            }
        }
    </table>
}
else
{
    <div class="text-center">No Unshipped Orders</div>
}

```

Each order is displayed with a Ship button that submits a form to the MarkShipped action method. We specified a different layout for the List view using the Layout property, which overrides the layout specified in the _ViewStart.cshtml file.

To add the layout, we use the MVC View Layout Page item template to create a file called _AdminLayout.cshtml in the Views/Shared folder and added the markup shown in Listing 80.

Listing 80. The Contents of the _AdminLayout.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>@ViewBag.Title</title>

```



```

</head>
<body class="panel panel-default">
  <div class="panel-heading"><h4>@ViewBag.Title</h4></div>
  <div class="panel-body">
    @RenderBody()
  </div>
</body>
</html>

```

To see and manage the orders in the application, start the application, select some products, and then check out. Then navigate to the /Order/List URL and you will see a summary of the order you created, as shown in Figure 63. Click the Ship button; the database will be updated, and the list of pending orders will be empty, see Figure 64. At the moment, there is nothing to stop customers from requesting the /Order/List URL and administering their own orders. We will restrict access to action methods later.

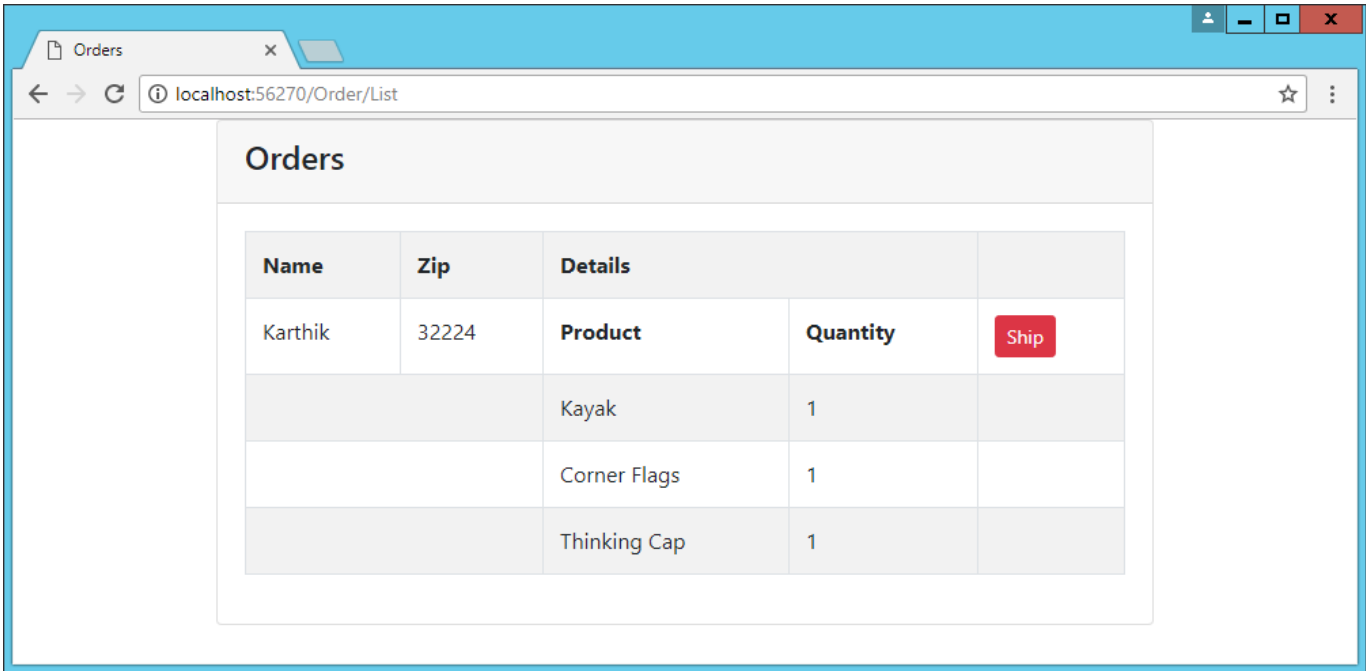


Figure 63. Managing orders

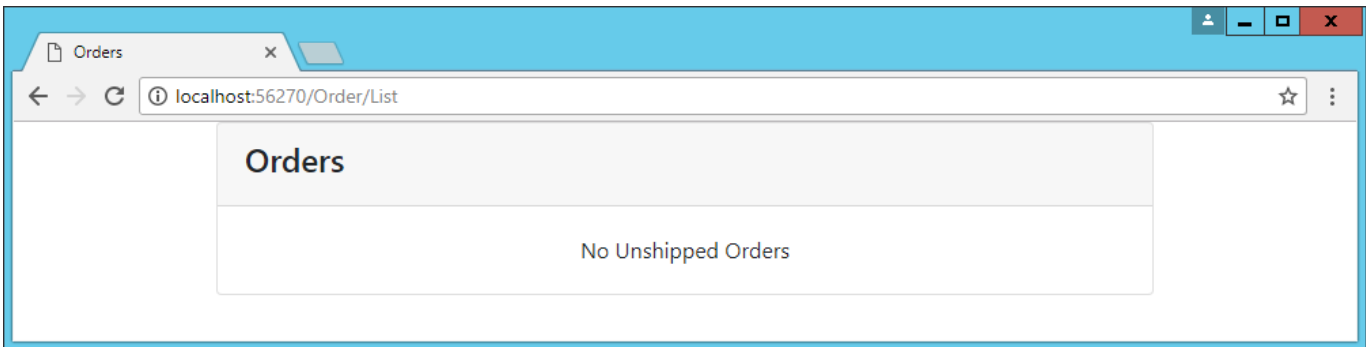


Figure 64. Display for empty pending orders

1.57 Task 52: Adding Catalog Management

The convention for managing more complex collections of items is to present the user with two types of pages: a list page and an edit page, as shown in Figure 65.

Item	Actions
Kayak	Edit Delete
Lifejacket	Edit Delete
Soccer ball	Edit Delete

Add New Item

Edit Item: Kayak

Name:

Description:

Category:

Price (\$):

Save Cancel

Figure 65. Sketch of a CRUD UI for the product catalog

Together, these pages allow a user to create, read, update, and delete items in the collection. Collectively, these actions are known as CRUD. Developers need to implement CRUD so often that Visual Studio scaffolding includes scenarios for creating CRUD controllers with predefined action methods. But like all the Visual Studio templates, it is better to learn how to use the features of the ASP.NET Core MVC directly.

Creating a CRUD Controller

We are going to start by creating a separate controller for managing the product catalog. We add a class file called `AdminController.cs` to the `Controllers` folder and use the code shown in Listing 81.

Listing 81. The Contents of the `AdminController.cs` File

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

namespace SportsStore.Controllers
{
    public class AdminController : Controller
    {
        private IProductRepository repository;

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

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

The controller constructor declares a dependency on the `IProductRepository` interface, which will be resolved when instances are created. The controller defines a single action method, `Index`, that calls the

View method to select the default view for the action, passing the set of products in the database as the view model.

Implementing the List View

The next step is to add a view for the Index action method of the Admin controller. We create the Views/Admin folder and added a Razor file called Index.cshtml, the contents of which are shown in Listing 82.

Listing 82. The Contents of the Index.cshtml File in the Views/Admin Folder

```
@model IEnumerable<Product>

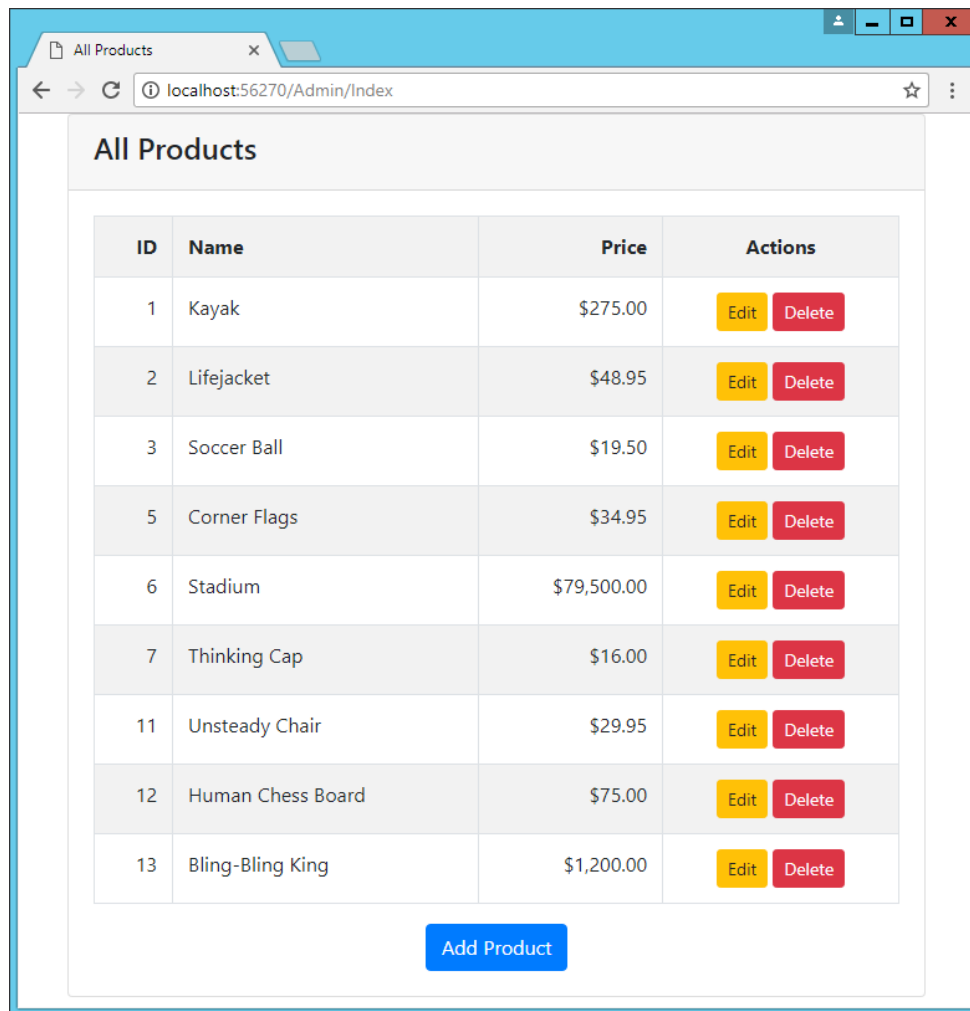
@{
    ViewBag.Title = "All Products";
    Layout = "_AdminLayout";
}

<table class="table table-striped table-bordered table-condensed">
    <tr>
        <th class="text-right">ID</th>
        <th>Name</th>
        <th class="text-right">Price</th>
        <th class="text-center">Actions</th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td class="text-right">@item.ProductID</td>
            <td>@item.Name</td>
            <td class="text-right">@item.Price.ToString("c")</td>
            <td class="text-center">
                <form asp-action="Delete" method="post">
                    <a asp-action="Edit" class="btn btn-sm btn-warning"
                       asp-route-productId="@item.ProductID">
                        Edit
                    </a>
                    <input type="hidden" name="ProductID" value="@item.ProductID" />
                    <button type="submit" class="btn btn-danger btn-sm">
                        Delete
                    </button>
                </form>
            </td>
        </tr>
    }
</table>

<div class="text-center">
    <a asp-action="Create" class="btn btn-primary">Add Product</a>
</div>
```

This view contains a table that has a row for each product with cells that contains the name of the table, the price, and buttons that will allow the product to be edited or deleted by sending requests to Edit and Delete actions. In addition to the table, there is an Add Product button that targets the Create action. We will add the Edit, Delete, and Create actions in the sections that follow, but you can see how the products are displayed by starting the application and requesting the /Admin/Index URL, as shown in Figure 66.

The Edit button is inside the form element so that the two buttons sit next to each other, working around the spacing that Bootstrap applies. The Edit button will send an HTTP GET request to the server in order to get the current details of a product; this doesn't require a form element. However, since the Delete button will make a change to the application state, we need to use an HTTP POST request—and that does require the form element.



The screenshot shows a web browser window with the address bar at `localhost:56270/Admin/Index`. The page title is "All Products". Below the title is a table with the following data:

ID	Name	Price	Actions
1	Kayak	\$275.00	Edit Delete
2	Lifejacket	\$48.95	Edit Delete
3	Soccer Ball	\$19.50	Edit Delete
5	Corner Flags	\$34.95	Edit Delete
6	Stadium	\$79,500.00	Edit Delete
7	Thinking Cap	\$16.00	Edit Delete
11	Unsteady Chair	\$29.95	Edit Delete
12	Human Chess Board	\$75.00	Edit Delete
13	Bling-Bling King	\$1,200.00	Edit Delete

Below the table is a blue button labeled "Add Product".

Figure 66. Displaying the list of products

1.58 Task 53: Editing Products

To provide create and update features, we will add a product-editing page like the one shown in Figure 65. These are the two parts to this job:

- Display a page that will allow the administrator to change values for the properties of a product
- Add an action method that can process those changes when they are submitted

Creating the Edit Action Method

Listing 83 shows the Edit action method we need to add to the Admin controller, which will receive the HTTP request sent by the browser when the user clicks an Edit button.

Listing 83. Adding an Edit Action Method in the AdminController.cs File

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

namespace SportsStore.Controllers
{
    public class AdminController : Controller
    {
        private IProductRepository repository;

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

        public ViewResult Index() => View(repository.Products);

        public ViewResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == productId));
    }
}
```

This simple method finds the product with the ID that corresponds to the productid parameter and passes it as a view model object to the View method.

Creating the Edit View

Now that we have an action method, we can create a view for it to display. Add a Razor view file called Edit.cshtml to the Views/Admin folder and added the markup shown in Listing 113.

Listing 84. The Contents of the Edit.cshtml File in the Views/Admin Folder

```
@model Product

@{
    ViewBag.Title = "Edit Product";
    Layout = "_AdminLayout";
}

<form asp-action="Edit" method="post">
    <input type="hidden" asp-for="ProductID" />
    <div class="form-group">
        <label asp-for="Name"></label>
        <input asp-for="Name" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Description"></label>
        <textarea asp-for="Description" class="form-control"></textarea>
    </div>
    <div class="form-group">
        <label asp-for="Category"></label>
        <input asp-for="Category" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
```

```

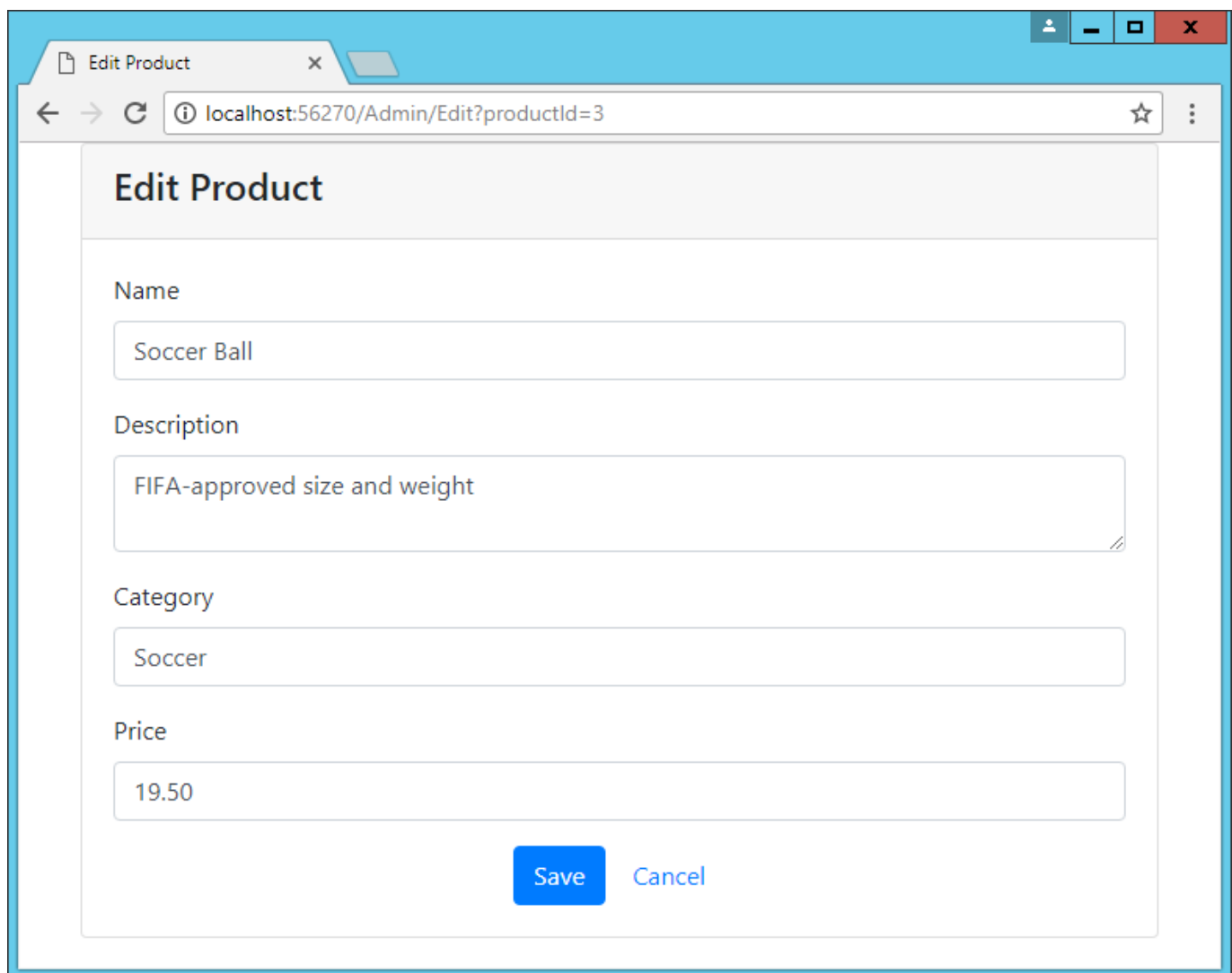
        <input asp-for="Price" class="form-control" />
    </div>
    <div class="text-center">
        <button class="btn btn-primary" type="submit">Save</button>
        <a asp-action="Index" class="btn btn-default">Cancel</a>
    </div>
</form>

```

The view contains an HTML form that uses tag helpers to generate much of the content, including setting the target for the form and a elements, setting the content of the label elements, and producing the name, id, and value attributes for the input and textarea elements.

We have used a hidden input element for the ProductID property for simplicity. The value of the ProductID is generated by the database as a primary key when a new object is stored by the Entity Framework Core, and safely changing it can be a complex process.

You can see the HTML produced by the view by starting the application, navigating to the /Admin/Index URL, and clicking the Edit button for one of the products, as shown in Figure 67.



The screenshot shows a web browser window with the title 'Edit Product'. The address bar shows the URL 'localhost:56270/Admin/Edit?productId=3'. The form is titled 'Edit Product' and contains the following fields:

- Name:** A text input field containing 'Soccer Ball'.
- Description:** A text area containing 'FIFA-approved size and weight'.
- Category:** A text input field containing 'Soccer'.
- Price:** A text input field containing '19.50'.

At the bottom of the form are two buttons: 'Save' (a blue button) and 'Cancel' (a light blue button).

Figure 67. Displaying product values for editing

1.59 Task 54: Updating the Product Repository

Before we can process edits, we need to enhance the product repository so that it is able to save changes. First, we need to add a new method to the `IProductRepository` interface, as shown in Listing 85.

Listing 85. Adding a Method to the `IProductRepository.cs` File

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

namespace SportsStore.Models
{
    public interface IProductRepository
    {
        IEnumerable<Product> Products { get; }

        void SaveProduct(Product product);
    }
}
```

We can now add the new method to the Entity Framework Core implementation of the repository, which is defined in the `EFProductRepository.cs` file, as shown in Listing 115.

Listing 86. Implementing the `SaveProduct` Method in the `EFProductRepository.cs` File

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

namespace SportsStore.Models
{
    public class EFProductRepository : IProductRepository
    {
        private ApplicationDbContext context;
        public EFProductRepository(ApplicationDbContext ctx)
        {
            context = ctx;
        }

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

        public void SaveProduct(Product product)
        {
            if (product.ProductID == 0)
            {
                context.Products.Add(product);
            }
            else
            {
                Product dbEntry = context.Products
                    .FirstOrDefault(p => p.ProductID == product.ProductID);
                if (dbEntry != null)
                {
                    dbEntry.Name = product.Name;
                }
            }
        }
    }
}
```

```

        dbEntry.Description = product.Description;
        dbEntry.Price = product.Price;
        dbEntry.Category = product.Category;
    }
}
context.SaveChanges();
}
}

```

The implementation of the `SaveChanges` method adds a product to the repository if the `ProductID` is 0; otherwise, it applies any changes to the existing entry in the database. We perform an update when we receive a `Product` parameter that has a `ProductID` that is not zero. We do this by getting a `Product` object from the repository with the same `ProductID` and updating each of the properties so they match the parameter object.

We can do this because the Entity Framework Core keeps track of the objects that it creates from the database. The object passed to the `SaveChanges` method is created by the MVC model binding feature, which means that the Entity Framework Core does not know anything about the new `Product` object and will not apply an update to the database when it is modified. There are lots of ways of resolving this issue, and we have taken the simplest one, which is to locate the corresponding object that the Entity Framework Core does know about and update it explicitly.

The addition of a new method in the `IProductRepository` interface has broken the fake repository class—`FakeProductRepository`—that we created in a previous session. We used the fake repository to kick-start the development process and demonstrate how services can be used to seamlessly replace interface implementations without needing to modify the components that rely on them. We don't need the fake repository any further, and in Listing 87, you can see that we remove the interface from the class declaration so that we don't have to keep modifying the class as we add repository features.

Listing 87. Disconnecting a Class from an Interface in the `FakeProductRepository.cs` File

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

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 }
        };
    }
}

```

1.60 Task 55: Handling Edit POST Requests

We are ready to implement an overload of the `Edit` action method in the `Admin` controller that will handle POST requests when the administrator clicks the `Save` button. Listing 88 shows the new action method.

Listing 88. Defining the POST-Handling Edit Action Method in the AdminController.cs File

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

namespace SportsStore.Controllers
{
    public class AdminController : Controller
    {
        private IProductRepository repository;

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

        public ViewResult Index() => View(repository.Products);

        public ViewResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == productId));

        [HttpPost]
        public IActionResult Edit(Product product)
        {
            if (ModelState.IsValid)
            {
                repository.SaveProduct(product);
                TempData["message"] = $"{product.Name} has been saved";
                return RedirectToAction("Index");
            }
            else
            {
                // there is something wrong with the data values
                return View(product);
            }
        }
    }
}
```

We check that the model binding process has been able to validate the data submitted to the user by reading the value of the `ModelState.IsValid` property. If everything is OK, we save the changes to the repository and redirect the user to the Index action so they see the modified list of products. If there is a problem with the data, we render the default view again so that the user can make corrections.

After we saved the changes in the repository, we store a message using the temp data feature, which is part of the ASP.NET Core session state feature. This is a key/value dictionary similar to the session data and view bag features we used previously. The key difference from session data is that temp data persists until it is read.

We cannot use ViewBag in this situation because ViewBag passes data between the controller and view and it cannot hold data for longer than the current HTTP request. When an edit succeeds, the browser is redirected to a new URL, so the ViewBag data is lost. We could use the session data feature, but then the message would be persistent until we explicitly removed it, which we would rather not have to do.

So, the temp data feature is the perfect fit. The data is restricted to a single user's session (so that users do not see each other's TempData) and will persist long enough for me to read it. We will read the data in the view rendered by the action method to which we have redirected the user, which we define in the next section.

Displaying a Confirmation Message

We are going to deal with the message we stored using TempData in the `_AdminLayout.cshtml` layout file, as shown in Listing 89. By handling the message in the template, we can create messages in any view that uses the template without needing to create additional Razor expressions.

Listing 89. Handling the ViewBag Message in the `_AdminLayout.cshtml` File

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div class="container">
        <div class="card">
            <div class="card-header">
                <h4>@ViewBag.Title</h4>
            </div>
            <div class="card-body">
                @if (TempData["message"] != null)
                {
                    <div class="alert alert-success">@TempData["message"]</div>
                }

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

The benefit of dealing with the message in the template like this is that users will see it displayed on whatever page is rendered after they have saved a change. At the moment, we return them to the list of products, but we could change the workflow to render some other view, and the users will still see the message (as long as the next view also uses the same layout).

We now have all the pieces in place to edit products. To see how it all works, start the application, navigate to the `/Admin/Index` URL, click the Edit button, and make a change. Click the Save button. You will be redirected to the `/Admin/Index` URL, and the TempData message will be displayed, as shown in Figure 68. The message will disappear if you reload the product list screen, because TempData is deleted when it is read. That is convenient since we do not want old messages hanging around.

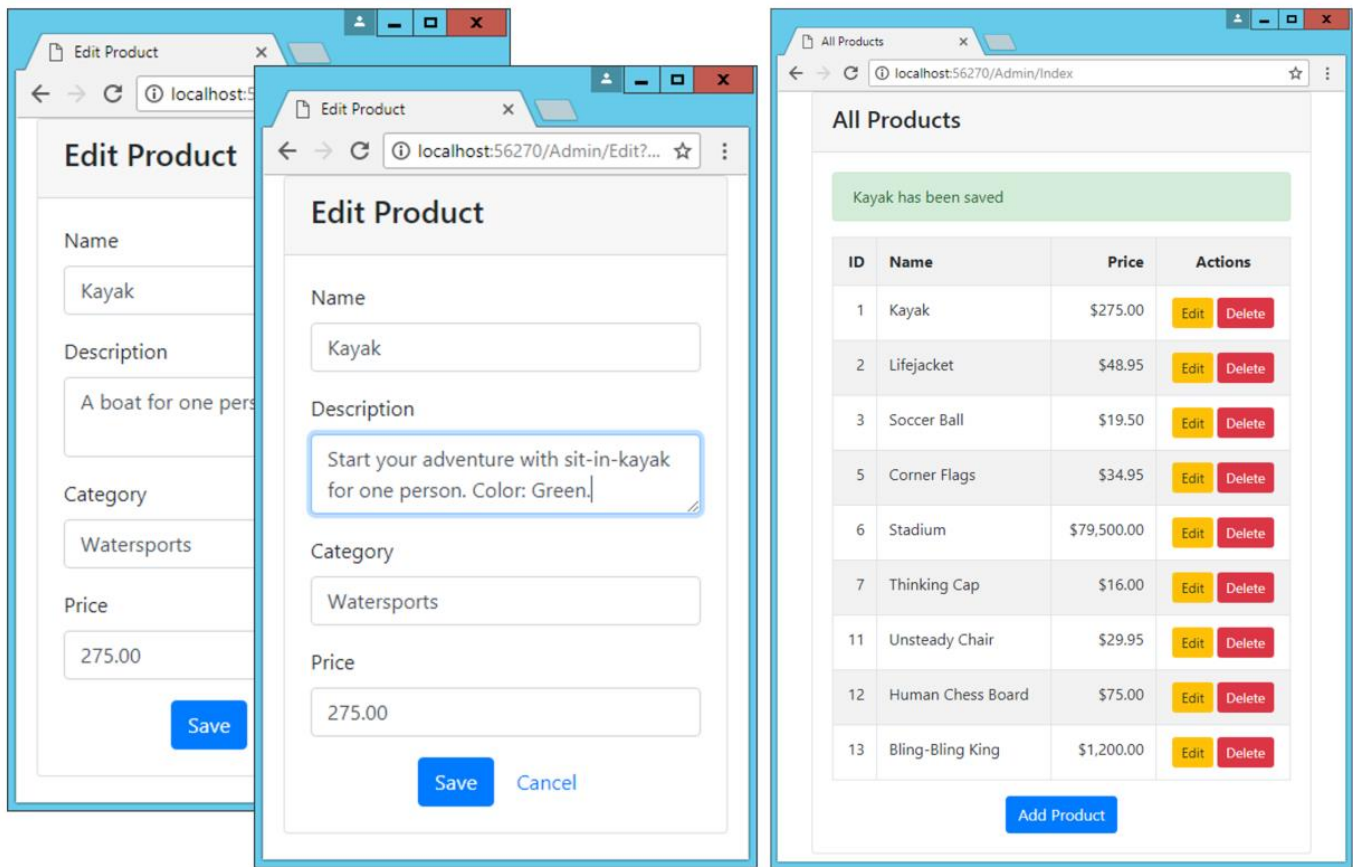


Figure 68. Editing a product and seeing the TempData message

1.61 Task 56: Adding Model Validation

We have reached the point where we need to add validation rules to the model classes. At the moment, the administrator could enter negative prices or blank descriptions, and SportsStore would happily store that data in the database. Whether or not the bad data would be successfully persisted would depend on whether it conformed to the constraints in the SQL tables created by Entity Framework Core, and that is not enough protection for most applications. To guard against bad data values, we decorate the properties of the Product class with attributes, as shown in Listing 90, just as we did for the Order class.

Listing 90. Applying Validation Attributes in the Product.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models
{
    public class Product
    {
        public int ProductID { get; set; }
```

```
[Required(ErrorMessage = "Please enter a product name")]
```

```

        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter a description")]
        public string Description { get; set; }

        [Required]
        [Range(0.01, double.MaxValue, ErrorMessage = "Please enter a positive price")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "Please specify a category")]
        public string Category { get; set; }
    }
}

```

In regards to client-side data input error checking for the Order class, we used a tag helper to display a summary of validation errors at the top of the form. For Product class, we are going to use a similar approach, but we are going to display error messages next to individual form elements in the Edit view in the Views/Admin folder, as shown in Listing 91.

Listing 91. Adding Validation Error Elements in the Edit.cshtml File

```

@model Product

@{
    ViewBag.Title = "Edit Product";
    Layout = "_AdminLayout";
}

<form asp-action="Edit" method="post">
    <input type="hidden" asp-for="ProductID" />
    <div class="form-group">
        <label asp-for="Name"></label>
        <div><span asp-validation-for="Name" class="text-danger"></span></div>
        <input asp-for="Name" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Description"></label>
        <div><span asp-validation-for="Description" class="text-danger"></span></div>
        <textarea asp-for="Description" class="form-control"></textarea>
    </div>
    <div class="form-group">
        <label asp-for="Category"></label>
        <div><span asp-validation-for="Category" class="text-danger"></span></div>
        <input asp-for="Category" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Price"></label>
        <div><span asp-validation-for="Price" class="text-danger"></span></div>
        <input asp-for="Price" class="form-control" />
    </div>
    <div class="text-center">
        <button class="btn btn-primary" type="submit">Save</button>
        <a asp-action="Index" class="btn btn-default">Cancel</a>
    </div>
</form>

```

When applied to a span element, the asp-validation-for attribute applies a tag helper that will add a validation error message for the specified property if there are any validation problems.

The tag helpers will insert an error message into the span element and add the element to the input-validation-error class, which makes it easy to apply CSS styles to error message elements, as shown in Listing 92.

Listing 92. Adding CSS to the _AdminLayout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
  <style>
    .input-validation-error {
      border-color: red;
      background-color: #fee;
    }
  </style>
</head>
<body class="panel panel-default">
  <div class="panel-heading"><h4>@ViewBag.Title</h4></div>
  <div class="panel-body">
    @if (TempData["message"] != null)
    {
      <div class="alert alert-success">@TempData["message"]</div>
    }
    @RenderBody()
  </div>
</body>
</html>
```

The CSS style we defined selects elements that are members of the input-validation-error class and applies a red border and background color.

You can apply the validation message tag helpers anywhere in the view, but it is conventional (and sensible) to put it somewhere near the problem element to give the user some context. Figure 69 shows the validation messages and cues that are displayed, which you can see by running the application, editing a product, and submitting invalid data.

The screenshot shows a web browser window with the title 'Edit Product' and the address bar displaying 'localhost:56270/Admin/Edit'. The form itself has a header 'Edit Product' and four input fields, each with a validation error message in red text:

- Name:** 'Please enter a product name' (above an empty text input field).
- Description:** 'Please enter a description' (above an empty text area).
- Category:** 'Please specify a category' (above an empty text input field).
- Price:** 'The value " " is invalid.' (above an empty text input field).

At the bottom of the form are two buttons: 'Save' (blue) and 'Cancel' (light blue).

Figure 69. Data validation when editing products

1.62 Task 57: Enabling Client-Side Validation

Currently, data validation is applied only when the administration user submits edits to the server, but most users expect immediate feedback if there are problems with the data they have entered. This is why developers often want to perform client-side validation, where the data is checked in the browser using JavaScript. MVC applications can perform client-side validation based on the data annotations we apply to the domain model class.

The first step is to add the JavaScript libraries that provide the client-side feature to the application, which is done in the `bower.json` file, as shown in Listing 93.

Listing 93. Adding JavaScript Packages in the `bower.json` File

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.7",
    "fontawesome": "4.7.0",
    "jquery": "3.2.1",
    "jquery-validation": "1.17.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

Client-side validation is built on top of the popular jQuery library, which simplifies working with the browser's DOM API. The next step is to add the JavaScript files to the layout so they are loaded when the SportsStore administration features are used, as shown in Listing 94.

Listing 94. Adding the Client-Side Validation Libraries to the `_AdminLayout.cshtml` File

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link rel="stylesheet" asp-href-include="lib/bootstrap/dist/css/*.min.css" />
  <title>@ViewBag.Title</title>
  <style>
    .input-validation-error {
      border-color: red;
      background-color: #fee;
    }
  </style>
  <script asp-src-include="lib/jquery/**/jquery.min.js"></script>
  <script asp-src-include="lib/jquery-validation/**/jquery.validate.min.js"></script>
  <script asp-src-include="lib/jquery-validation-unobtrusive/**/**.min.js"></script>
</head>
<body>
  <div class="container">
    <div class="card">
      <div class="card-header">
        <h4>@ViewBag.Title</h4>
      </div>
      <div class="card-body">
        @if (TempData["message"] != null)
        {
          <div class="alert alert-success">@TempData["message"]</div>
        }

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

These additions use a tag helper to select the files that are included in the script elements. Use of wildcards to select JavaScript files means that the application won't break if the names of the files in the Bower package change when a new version is released. Some caution is required, however, because it is easy to select files that you didn't expect.

Enabling client-side validation doesn't cause any visual change, but the constraints specified by the attributes applied to the C# model class are enforced at the browser, preventing the user from submitting the form with bad data and providing immediate feedback when there is a problem.

Note: Client-side Validation Caution

In general, using client-side data validation is a good idea. It offloads some of the work from your server and gives users immediate feedback about the data they are providing. However, you should not be tempted to perform authentication at the client, as this would typically involve sending valid credentials to the client so they can be used to check the username and password that the user has entered, or at least trusting the client's report of whether they have successfully authenticated. Authentication should always be done at the server.

Therefore, it is good to validate data entered by users at client-side, but do not authenticate in client-side, authentication should be done at server-side.

1.63 Task 58: Creating New Products

Next, we will implement the Create action method, which is the one specified by the Add Product link in the main product list page. This will allow the administrator to add new items to the product catalog. Adding the ability to create new products will require one small addition to the application. This is a great example of the power and flexibility of a well structured MVC application. First, add the Create method, shown in Listing 95, to the Admin controller.

Listing 95. Adding the Create Action Method to the AdminController.cs File

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

namespace SportsStore.Controllers
{
    public class AdminController : Controller
    {
        private IProductRepository repository;

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

        public IActionResult Index() => View(repository.Products);

        public IActionResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == productId));

        [HttpPost]
```



```

public IActionResult Edit(Product product)
{
    if (ModelState.IsValid)
    {
        repository.SaveProduct(product);
        TempData["message"] = $"{product.Name} has been saved";
        return RedirectToAction("Index");
    }
    else
    {
        // there is something wrong with the data values
        return View(product);
    }
}

public ViewResult Create() => View("Edit", new Product());
}

```

The Create method does not render its default view. Instead, it specifies that the Edit view should be used. It is perfectly acceptable for one action method to use a view that is usually associated with another view. In this case, we provide a new Product object as the view model so that the Edit view is populated with empty fields. That is the only change that is required because the Edit action method is already set up to receive Product objects from the model binding system and store them in the database.

You can test this functionality by starting the application, navigating to /Admin/Index, clicking the Add Product button, and populating and submitting the form. The details you specify in the form will be used to create a new product in the database, which will then appear in the list, as shown in Figure 70.

Removing ID values from Catalog Display

As we use auto numbering in database for Product ID, depending upon database settings and number of times we add and delete products, ID numbering will get out of sequence and thus might face discrepancies like in the Figure 70. Thus, displaying ID that are autonumbering based is generally not advised. We can edit the Index.cshtml View file in the View/Admin folder to not display ID header and Product ID values, as indicated in the Listing 96. Figure 71 shows updated catalog view.

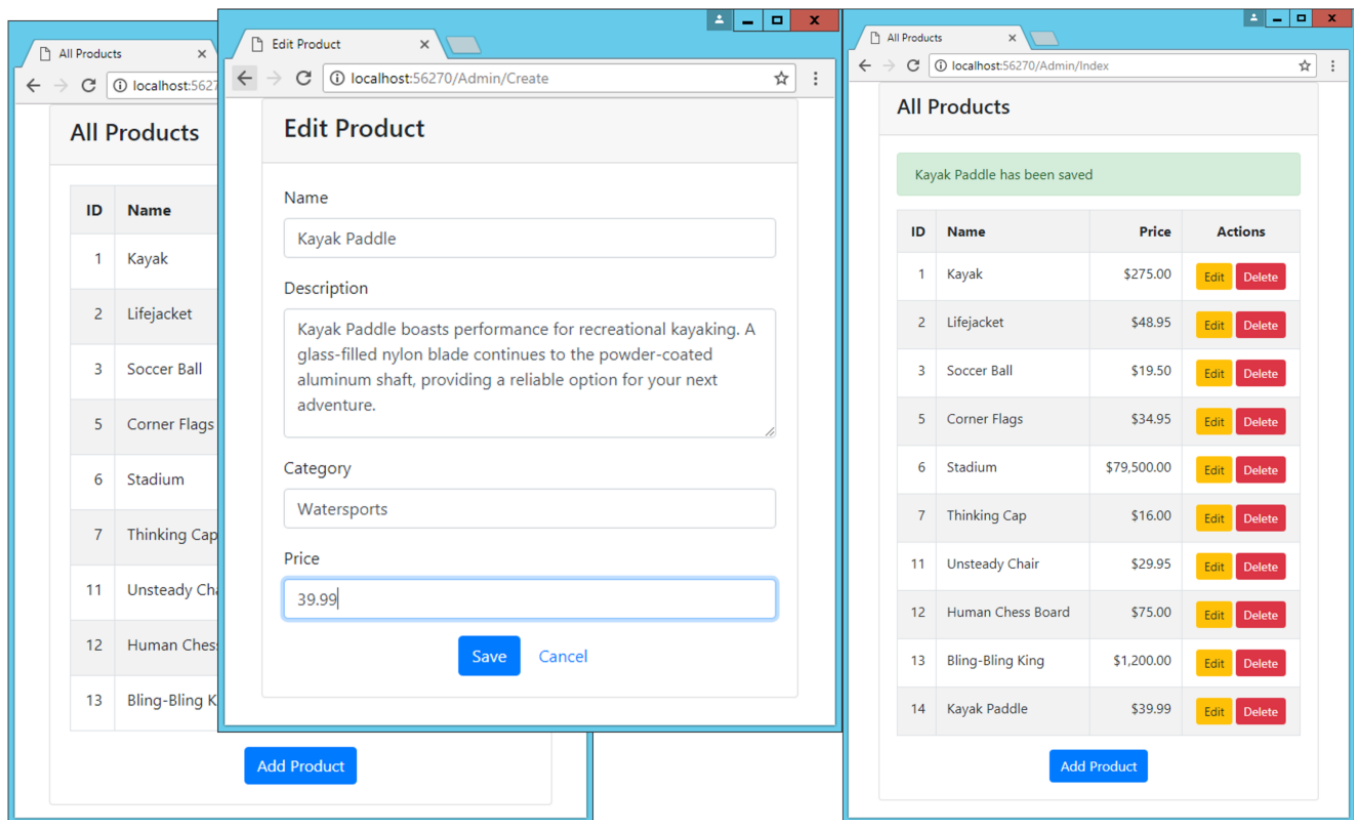


Figure 70. Adding a new product to the catalog

Listing 96. Editing Index.cshtml in the View/Admin Folder

```
@model IEnumerable<Product>
```

```
@{
    ViewBag.Title = "All Products";
    Layout = "_AdminLayout";
}
```

```
<table class="table table-striped table-bordered table-condensed">
    <tr>
```

```
        <!--<th class="text-right">ID</th-->
```

```
        <th>Name</th>
```

```
        <th class="text-right">Price</th>
```

```
        <th class="text-center">Actions</th>
```

```
    </tr>
```

```
    @foreach (var item in Model)
```

```
    {
```

```
        <tr>
```

```
            <!--<td class="text-right">@item.ProductID</td-->
```

```
            <td>@item.Name</td>
```

```
            <td class="text-right">@item.Price.ToString("c")</td>
```

```
            <td class="text-center">
```

```
                <form asp-action="Delete" method="post">
```

```
                    <a asp-action="Edit" class="btn btn-sm btn-warning"
```

```
                        asp-route-productId="@item.ProductID">
```

```
                        Edit
```

```
                    </a>
```

```
                    <input type="hidden" name="ProductID" value="@item.ProductID" />
```

```

        <button type="submit" class="btn btn-danger btn-sm">
            Delete
        </button>
    </form>
</td>
</tr>
}
</table>

<div class="text-center">
    <a asp-action="Create" class="btn btn-primary">Add Product</a>
</div>

```

Name	Price	Actions
Kayak	\$275.00	Edit Delete
Lifejacket	\$48.95	Edit Delete
Soccer Ball	\$19.50	Edit Delete
Corner Flags	\$34.95	Edit Delete
Stadium	\$79,500.00	Edit Delete
Thinking Cap	\$16.00	Edit Delete
Unsteady Chair	\$29.95	Edit Delete
Human Chess Board	\$75.00	Edit Delete
Bling-Bling King	\$1,200.00	Edit Delete
Kayak Paddle	\$39.99	Edit Delete

[Add Product](#)

Figure 71. Updated Product Catalog View

1.64 Task 59: Deleting Products

Adding support for deleting items is also simple. First, we add a new method to the `IProductRepository` interface, as shown in Listing 97.

Listing 97. Adding a Method to Delete Products to the `IProductRepository.cs` File

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

namespace SportsStore.Models
{
    public interface IProductRepository
    {
        IEnumerable<Product> Products { get; }

        void SaveProduct(Product product);

        Product DeleteProduct(int productID);
    }
}
```

Next, we implement this method in the Entity Framework Core repository class, `EFProductRepository`, as shown in Listing 98.

Listing 98. Implementing Deletion Support in the `EFProductRepository.cs` File

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

namespace SportsStore.Models
{
    public class EFProductRepository : IProductRepository
    {
        private ApplicationDbContext context;
        public EFProductRepository(ApplicationDbContext ctx)
        {
            context = ctx;
        }

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

        public void SaveProduct(Product product)
        {
            if (product.ProductID == 0)
            {
                context.Products.Add(product);
            }
            else
            {
                Product dbEntry = context.Products
                    .FirstOrDefault(p => p.ProductID == product.ProductID);
                if (dbEntry != null)
                {

```

```

        dbEntry.Name = product.Name;
        dbEntry.Description = product.Description;
        dbEntry.Price = product.Price;
        dbEntry.Category = product.Category;
    }
}
context.SaveChanges();
}

public Product DeleteProduct(int productID)
{
    Product dbEntry = context.Products
        .FirstOrDefault(p => p.ProductID == productID);
    if (dbEntry != null)
    {
        context.Products.Remove(dbEntry);
        context.SaveChanges();
    }
    return dbEntry;
}
}
}

```

The final step is to implement a Delete action method in the Admin controller. This action method should support only POST requests because deleting objects is not an idempotent operation. Browsers and caches are free to make GET requests without the user's explicit consent, so we must be careful to avoid making changes as a consequence of GET requests. Listing 99 shows the new action method.

Listing 99. Adding the Delete Action Method in the AdminController.cs File

```

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

namespace SportsStore.Controllers
{
    public class AdminController : Controller
    {
        private IProductRepository repository;

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

        public ViewResult Index() => View(repository.Products);

        public ViewResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == productId));

        [HttpPost]
        public IActionResult Edit(Product product)
        {
            if (ModelState.IsValid)
            {
                repository.SaveProduct(product);
                TempData["message"] = $"{product.Name} has been saved";
                return RedirectToAction("Index");
            }
        }
    }
}

```

```

    }
    else
    {
        // there is something wrong with the data values
        return View(product);
    }
}

```

```

public ViewResult Create() => View("Edit", new Product());

```

```

[HttpPost]
public IActionResult Delete(int productId)
{
    Product deletedProduct = repository.DeleteProduct(productId);
    if (deletedProduct != null)
    {
        TempData["message"] = $"{deletedProduct.Name} was deleted";
    }
    return RedirectToAction("Index");
}

```

You can see the delete feature by starting the application, navigating to /Admin/Index, and clicking one of the Delete buttons in the product list page, as shown in Figure 72. As shown in the figure, we have taken advantage of the TempData variable to display a message when a product is deleted from the catalog.

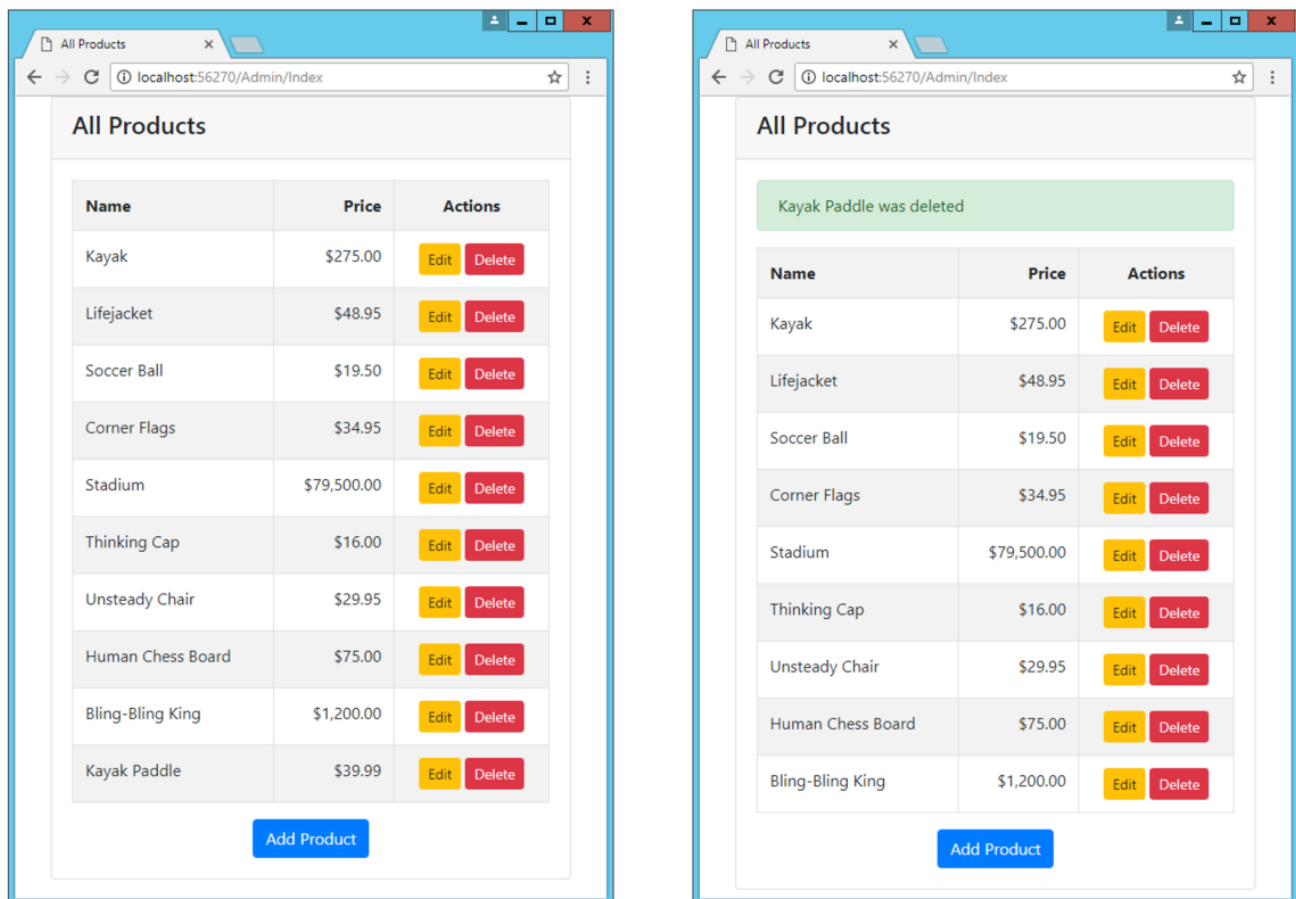


Figure 72. Deleting a product from the catalog

1.65 Securing the Administration Features

We introduced the administration capability and showed you how to implement CRUD operations that allow the administrator to create, read, update, and delete products from the repository and mark orders as shipped. Any users can access the administration features using the /Admin/Index and /Order/List URLs. Thus, we need to secure the administration functions so that they are not available to all users.

Authentication and authorization are provided by the ASP.NET Core Identity system, which integrates neatly into both the ASP.NET Core platform and MVC applications. In the sections that follow, we will create a basic security setup that allows one user, called Admin, to authenticate and access the administration features in the application. ASP.NET Core Identity provides many more features for authenticating users and authorizing access to application features and data to create and manage user accounts, use roles and policies, and support authentication from third parties, such as Microsoft, Google, Facebook, and Twitter. In this session, however, our goal is just to get enough functionality in place to prevent customers from being able to access the sensitive parts of the SportsStore application and, in doing so, give you a flavor of how authentication and authorization fit into an MVC application.

1.66 Task 60: Create Application User class inheriting IdentityUser base class

Most of the online tutorials for ASP.NET Core Identity system uses separate database for managing individual user accounts. As we already have an existing database for managing application data, we will use the same database for managing user accounts using Identity system as well.

Before we begin setup for using Identity system, we need to define a class to represent users in our application. We will do that by creating an AppUser class in the Models folder and make sure that it inherits the IdentityUser base class from ASP.NET Core Identity. The IdentityUser base class contains all the properties and methods necessary from ASP.NET Core Identity. See Listing 100 for details.

Listing 100. Adding AppUser class to Models Folder

```
using Microsoft.AspNetCore.Identity;
using System;

namespace SportsStore.Models
{
    public class AppUser : IdentityUser<Guid>
    {
    }
}
```

1.67 Task 61: Modify ApplicationDbContext class

In order to use the ASP.NET Core Identity, we have to modify our existing ApplicationDbContext class in the Models folder. As mentioned earlier, instead of creating a separate IdentityDbContext, we are going to use our existing ApplicationDbContext class by changing the inheritance to IdentityDbContext so that we can take advantage of the data and behavior added by ASP.NET Core Identity. See Listing 101 for modifications to be made to ApplicationDbContext class.

Listing 101. Modify ApplicationDbContext class in Models Folder

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

namespace SportsStore.Models
{
    public class ApplicationDbContext : IdentityDbContext<AppUser, IdentityRole<Guid>, Guid>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) :
base(options) { }
        public DbSet<Product> Products { get; set; }
        public DbSet<Order> Orders { get; set; }
    }
}
```

Configuring the Application

Like other ASP.NET Core features, Identity is configured in the Startup class. Listing 102 shows the additions we need to make to set up Identity in the SportsStore project. In the ConfigureServices method, we used the AddIdentity method to set up the Identity services using the built-in classes to represent users and roles. In the Configure method, we called the UseAuthentication method to set up the components that will intercept requests and responses to implement the security policy.

Listing 102. Configuring Identity in the Startup.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;

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

        // This method gets called by the runtime. Use this method to add services to the
        container.
        // For more information on how to configure your application, visit
        https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
```



```

{
    services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
    services.AddTransient<IProductRepository, EFProductRepository>();
    services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
    services.AddTransient<IOrderRepository, EFOOrderRepository>();

    services.AddIdentity<AppUser, IdentityRole<Guid>>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();
    services.AddMemoryCache();
    services.AddSession();
}

// This method gets called by the runtime. Use this method to configure the HTTP request
pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseAuthentication();
    app.UseSession();
    app.UseMvc(routes => {

        routes.MapRoute(
            name: null,
            template: "{category}/Page{page:int}",
            defaults: new { controller = "Product", action = "List" }
        );
        routes.MapRoute(
            name: null,
            template: "Page{page:int}",
            defaults: new { controller = "Product", action = "List", page = 1 }
        );
        routes.MapRoute(
            name: null,
            template: "{category}",
            defaults: new { controller = "Product", action = "List", page = 1 }
        );
        routes.MapRoute(
            name: null,
            template: "",
            defaults: new { controller = "Product", action = "List", page = 1 });

        routes.MapRoute(name: null, template: "{controller}/{action}/{id?}");

    });
}
}
}

```

1.68 Task 62: Creating and Applying the Database Migration

All of the components are in place, and it is time to use the Entity Framework Core migrations feature to define the schema and apply it to the database. Open the Package Manager Console and run the following command to create the migration:

Add-Migration Identity

Once Entity Framework Core has generated the migration, run the following command to create the database and run the migration commands:

Update-Database

The result is a new set of tables created by the Identity system that you can inspect using the Visual Studio SQL Server Object Explorer, as shown in the Figure 73.

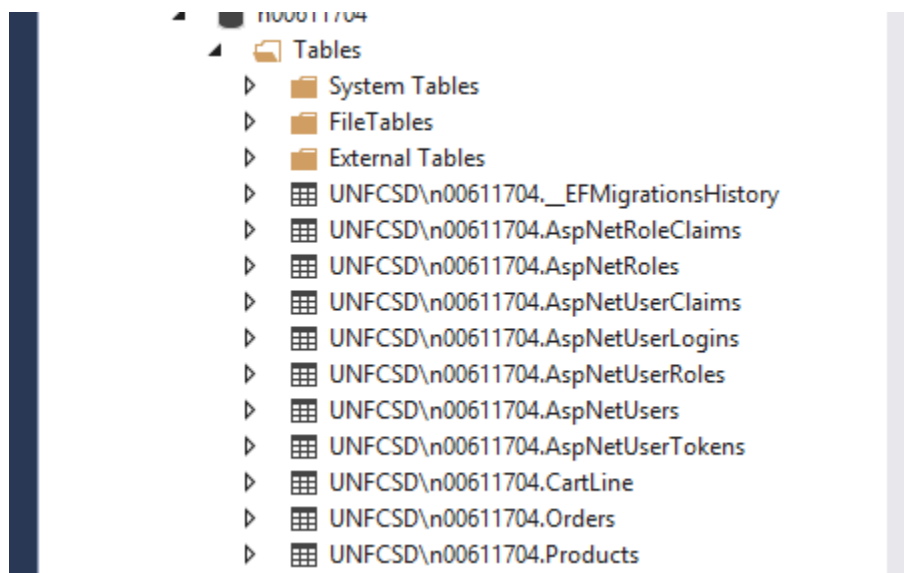


Figure 73. Deleting a product from the catalog

Applying a Basic Authorization Policy

Now that we have installed and configured ASP.NET Core Identity, we can apply an authorization policy to the parts of the application that we want to protect. We are going to use the most basic authorization policy possible, which is to allow access to any authenticated user. Although this can be a useful policy in real applications as well, there are also options for creating finergrained authorization controls, but since the SportsStore application has only one user, distinguishing between anonymous and authenticated requests is sufficient.

The Authorize attribute is used to restrict access to action methods, and in Listing 103, you can see that we use the attribute to protect access to the administration actions in the Order controller.

Listing 103. Restricting Access in the OrderController.cs File

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

namespace SportsStore.Controllers
{
    public class OrderController : Controller
    {
        private IOrderRepository repository;
        private Cart cart;
        public OrderController(IOrderRepository repoService, Cart cartService)
        {
            repository = repoService;
            cart = cartService;
        }

        [Authorize]
        public ViewResult List() => View(repository.Orders.Where(o => !o.Shipped));

        [HttpPost]
        [Authorize]
        public IActionResult MarkShipped(int orderID)
        {
            Order order = repository.Orders
                .FirstOrDefault(o => o.OrderID == orderID);
            if (order != null)
            {
                order.Shipped = true;
                repository.SaveOrder(order);
            }
            return RedirectToAction(nameof(List));
        }

        public ViewResult Checkout() => View(new Order());

        [HttpPost]
        public IActionResult Checkout(Order order)
        {
            if (cart.Lines.Count() == 0)
            {
                ModelState.AddModelError("", "Sorry, your cart is empty!");
            }
            if (ModelState.IsValid)
            {
                order.Lines = cart.Lines.ToArray();
                repository.SaveOrder(order);
                return RedirectToAction(nameof(Completed));
            }
            else
            {
                return View(order);
            }
        }

        public ViewResult Completed()
        {
            cart.Clear();
            return View();
        }
    }
}
```

```

    }
}
}

```

We don't want to stop unauthenticated users from accessing the other action methods in the Order controller, so we have applied the `Authorize` attribute only to the `List` and `MarkShipped` methods. We want to protect all of the action methods defined by the Admin controller, and we can do this by applying the `Authorize` attribute to the controller class, which then applies the authorization policy to all the action methods it contains, as shown in Listing 104.

Listing 104. Restricting Access in the AdminController.cs File

```

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

namespace SportsStore.Controllers
{
    [Authorize]
    public class AdminController : Controller
    {
        private IProductRepository repository;

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

        public IActionResult Index() => View(repository.Products);

        public IActionResult Edit(int productId) =>
            View(repository.Products
                .FirstOrDefault(p => p.ProductID == productId));

        [HttpPost]
        public IActionResult Edit(Product product)
        {
            if (ModelState.IsValid)
            {
                repository.SaveProduct(product);
                TempData["message"] = $"{product.Name} has been saved";
                return RedirectToAction("Index");
            }
            else
            {
                // there is something wrong with the data values
                return View(product);
            }
        }

        public IActionResult Create() => View("Edit", new Product());

        [HttpPost]
        public IActionResult Delete(int productId)
        {
            Product deletedProduct = repository.DeleteProduct(productId);
            if (deletedProduct != null)
            {

```

```

        TempData["message"] = $"{deletedProduct.Name} was deleted";
    }
    return RedirectToAction("Index");
}
}
}

```

When you run the application and try to access Admin/Index, now you will not be able to access it. You will get 404 error message as shown in the Figure 74, as we have not created ability for users to login, which we will create next.

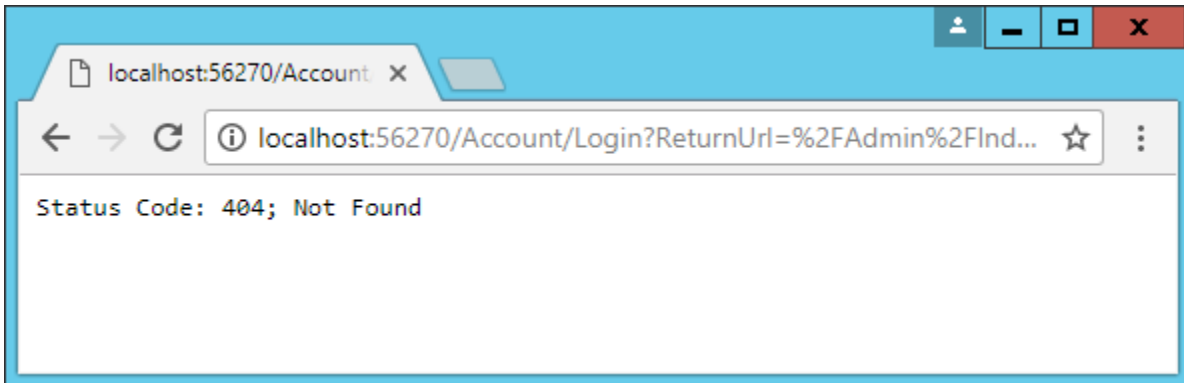


Figure 74. Source not found error message

1.69 Task 63: Creating the Account Controller

When an unauthenticated user sends a request that requires authorization, they are redirected to the /Account/Login URL, which the application can use to prompt the user for their credentials. In preparation, we add a view model to represent the user's credentials by adding a class file called LoginModel.cs to the Models/ViewModels folder and use it to define the class shown in Listing 105.

Listing 105. The Contents of the LoginModel.cs File in the Models/ViewModels Folder

```

using System.ComponentModel.DataAnnotations;

namespace SportsStore.Models.ViewModels
{
    public class LoginModel
    {
        public string UserName { get; set; }

        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [UIHint("password")]
        public string Password { get; set; }

        public string ReturnUrl { get; set; } = "/";
    }
}

```

The Email, and Password properties have been decorated with the Required attribute, which uses model validation to ensure that values have been provided. The Password property has been decorated with the UIHint attribute so that when we use the asp-for attribute on the input element in the login Razor view, the tag helper will set the type attribute to password; that way, the text entered by the user isn't visible onscreen.

Next, we add a class file called AccountController.cs to the Controllers folder and use it to define the controller shown in Listing 106. This is the controller that will respond to requests to the following URLs:
/Account/Login
/Account/Index
/Account/Create

Listing 106. The Contents of the AccountController.cs File in the Controllers Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;
using SportsStore.Models;

namespace SportsStore.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private UserManager<AppUser> userManager;
        private SignInManager<AppUser> signInManager;

        public AccountController(UserManager<AppUser> userMgr,
            SignInManager<AppUser> signInMgr)
        {
            userManager = userMgr;
            signInManager = signInMgr;
        }

        [AllowAnonymous]
        public ViewResult Login(string returnUrl)
        {
            return View(new LoginModel
            {
                ReturnUrl = returnUrl
            });
        }

        [HttpPost]
        [AllowAnonymous]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Login(LoginModel loginModel)
        {
            if (ModelState.IsValid)
            {
                AppUser user = await userManager.FindByEmailAsync(loginModel.Email);
```

```

        if (user != null)
        {
            await signInManager.SignOutAsync();
            if ((await signInManager.PasswordSignInAsync(user,
                loginModel.Password, false, false)).Succeeded)
            {
                return Redirect(loginModel?.ReturnUrl ?? "/Admin/Index");
            }
        }
    }
    ModelState.AddModelError("", "Invalid name or password");
    return View(loginModel);
}

public async Task<RedirectResult> Logout(string returnUrl = "/")
{
    await signInManager.SignOutAsync();
    return Redirect(returnUrl);
}

// GET: /<controller>/
[AllowAnonymous]
public IActionResult Index()
{
    return View(userManager.Users);
}

[AllowAnonymous]
public IActionResult Create()
{
    return View();
}

//POST: Account/Create
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(LoginModel model)
{
    if (ModelState.IsValid)
    {
        AppUser user = new AppUser
        {
            UserName = model.UserName,
            Email = model.Email
        };

        IdentityResult result = await userManager.CreateAsync(user, model.Password);

        if (result.Succeeded)
        {
            //await signInManager.SignInAsync(user, isPersistent: false);
            return RedirectToAction("Index");
        }
        else
        {
            foreach (IdentityError error in result.Errors)
            {
                ModelState.AddModelError("", error.Description);
            }
        }
    }
}

```

```

        }
    }
    }
    return View(model);
}

private void AddErrors(IdentityResult result)
{
    foreach (IdentityError error in result.Errors)
    {
        ModelState.TryAddModelError("", error.Description);
    }
}

public async Task<IActionResult> Delete(string id)
{
    AppUser user = await userManager.FindByIdAsync(id);

    if (user != null)
    {
        IdentityResult result = await userManager.DeleteAsync(user);
        if (result.Succeeded)
        {
            return RedirectToAction("Index");
        }
        else
        {
            AddErrors(result);
        }
    }
    else
    {
        ModelState.AddModelError("", "User Not Found");
    }

    return View("Index", userManager.Users);
}

public async Task<IActionResult> Edit(string id)
{
    AppUser user = await userManager.FindByIdAsync(id);

    if (user != null)
    {
        return View(user);
    }
    else
    {
        return RedirectToAction("Index");
    }
}
}
}

```


1.70 Task 64: Creating Login View

When the user is redirected to the /Account/Login URL, the GET version of the Login action method renders the default view for the page, providing a view model object that includes the URL that the browser should be redirected to if the authentication request is successful.

Authentication credentials are submitted to the POST version of the Login method, which uses the UserManager<AppUser> and SignInManager<AppUser> services that have been received through the controller's constructor to authenticate the user and log them into the system. If there is an authentication failure, then we create a model validation error and render the default view; however, if authentication is successful, then we redirect the user to the URL that they want to access before they are prompted for their credentials.

To provide the Login method with a view to render, we create the Views/Account folder and add a Razor view file called Login.cshtml with the contents shown in Listing 107.

Listing 107. The Contents of the Login.cshtml File in the Views/Account Folder

```
@model LoginModel

@{
    ViewBag.Title = "Log In";
    Layout = "_AdminLayout";
}

<div class="text-danger" asp-validation-summary="All"></div>

<form asp-action="Login" asp-controller="Account" method="post">
    <input type="hidden" asp-for="ReturnUrl" />
    <div class="form-group">
        <label asp-for="Email"></label>
        <div><span asp-validation-for="Email" class="text-danger"></span></div>
        <input asp-for="Email" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Password"></label>
        <div><span asp-validation-for="Password" class="text-danger"></span></div>
        <input asp-for="Password" class="form-control" />
    </div>
    <button class="btn btn-primary" type="submit">Log In</button>
</form>
```

The final step is a change to the shared administration layout to add a button that will log the current user out by sending a request to the Logout action, as shown in Listing 108. This is a useful feature that makes it easier to test the application, without which you would need to clear the browser's cookies in order to return to the unauthenticated state.

Listing 108. Adding a Logout Button in the _AdminLayout.cshtml File

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

```

<style>
    .input-validation-error {
        border-color: red;
        background-color: #fee;
    }
</style>
<script asp-src-include="lib/jquery/**/jquery.min.js"></script>
<script asp-src-include="lib/jquery-validation/**/jquery.validate.min.js"></script>
<script asp-src-include="lib/jquery-validation-unobtrusive/**/*.*.min.js"></script>

</head>
<body>
    <div class="container">
        <div class="card">
            <div class="card-header">
                <h4>@ViewBag.Title</h4>
                <a class="btn btn-sm btn-primary float-right" asp-action="Logout" asp-
controller="Account">Log Out</a>
            </div>
            <div class="card-body">
                @if (TempData["message"] != null)
                {
                    <div class="alert alert-success">@TempData["message"]</div>
                }

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

```

1.71 Task 65: Creating User Accounts

While we have created tables for user accounts using Identity system, we have not created any user accounts. Thus, we cannot log in and access admin features. In order to create user accounts, we need to add a Create Razor view to the Views/Account folder. You can see code for Create view in Listing 109. We have already added relevant methods for Create feature in the Account Controller class. See Task 106. This Create method will use HTTP Post to create a user record containing user name, email, and password. Create view now can be accessed using Account/Create URL.

Listing 109. The Contents for Create.cshtml File in the Views/Account Folder

```

@model LoginModel

@{
    ViewBag.Title = "Create User Account";
    Layout = "_AdminLayout";
}

<div asp-validation-summary="All" class="text-danger"></div>

<form method="post" asp-action="Create" asp-antiforgery="true" class="form-horizontal">
    <div class="form-group">
        <label class="control-label" asp-for="UserName">User Name:</label>
        <input class="form-control" asp-for="UserName" />
    </div>

```

```

</div>
<div class="form-group">
    <label class="control-label" asp-for="Email">Email:</label>
    <input class="form-control" asp-for="Email" />
</div>
<div class="form-group">
    <label class="control-label" asp-for="Password">Password:</label>
    <input class="form-control" asp-for="Password" type="password" />
</div>

<div class="form-group">
    <div class="col-sm-offset-2">
        <button type="submit" class="btn btn-primary">Create</button>
        <button type="reset" class="btn btn-primary">Reset</button>
        <a asp-action="Index" class="btn btn-primary">Cancel</a>
    </div>
</div>
</form>

```

1.72 Task 66: Listing User Accounts

In order to list existing user accounts, we need to add a Razor view called Index in the Views/Account folder. Use contents provided in Listing 110 for the Index.cshtml file. The Index view receive IEnumerable AppUsers from the Index method in the Account controller. From the listing it can be noted that index displays ID, user name, and email of the users in a tabular format. The Index view also provide us ability to edit and delete a user. Edit and Delete operations are supported by associated methods in the Account controller class. The Index view can be accessed using Account/Index URL.

Listing 110. The Contents for Index.cshtml File in the Views/Account Folder

```

@model IEnumerable<AppUser>

@{
    ViewBag.Title = "User Accounts";
    Layout = "_AdminLayout";
}

<div class="text-danger" asp-validation-summary="ModelOnly"></div>

<table class="table table-condensed table-bordered">
    <tr>
        <!--<th>ID</th>-->
        <th>UserName</th>
        <th>Email</th>
        <!--<th>Password</th>-->
    </tr>
    @if (Model.Count() == 0)
    {
        <tr>
            <td colspan="3" class="text-center">
                No User Accounts
            </td>
        </tr>
    }
    else
    {
        foreach (AppUser user in Model)
        {

```

```

<tr>
  <!--<td>@user.Id</td>-->
  <td>@user.UserName</td>
  <td>@user.Email</td>
  <!--<td>@user.PasswordHash</td>-->
  <td>
    <a class="btn btn-success" asp-controller="Account" asp-action="Edit" asp-
route-id="@user.Id">Edit</a>
    <a class="btn btn-danger" asp-controller="Account" asp-action="Delete" asp-
route-id="@user.Id">Delete</a>
  </td>
</tr>
}
</table>

<a class="btn btn-primary" asp-action="Create">Create</a>
<a class="btn btn-primary" asp-controller="Account" asp-action="Index">Home</a>

```

1.73 Task 67: Create a User Account

Everything is in place and you can test the security policy by starting with creating a user account via by requesting for the /Account/Create URL. Create a user account by providing User Name, Email, and Password as shown in the Figure 75. Remember email and password you type in as you need to them to authenticate your access to restricted features. You can view account create by accessing /Account/Index URL, see Figure 76.

The screenshot shows a web browser window with the title 'Create User Account'. The address bar shows 'localhost:56270/Account/Create'. The form contains the following elements:

- Title:** Create User Account
- Log Out Button:** A blue button in the top right corner.
- User Name:** A text input field containing 'Admin1'.
- Email:** A text input field containing 'admin1@cop3855.com'.
- Password:** A text input field with masked characters (dots) and a cursor at the end.
- Buttons:** Three blue buttons at the bottom: 'Create', 'Reset', and 'Cancel'.

Figure 75. Creating New User Account

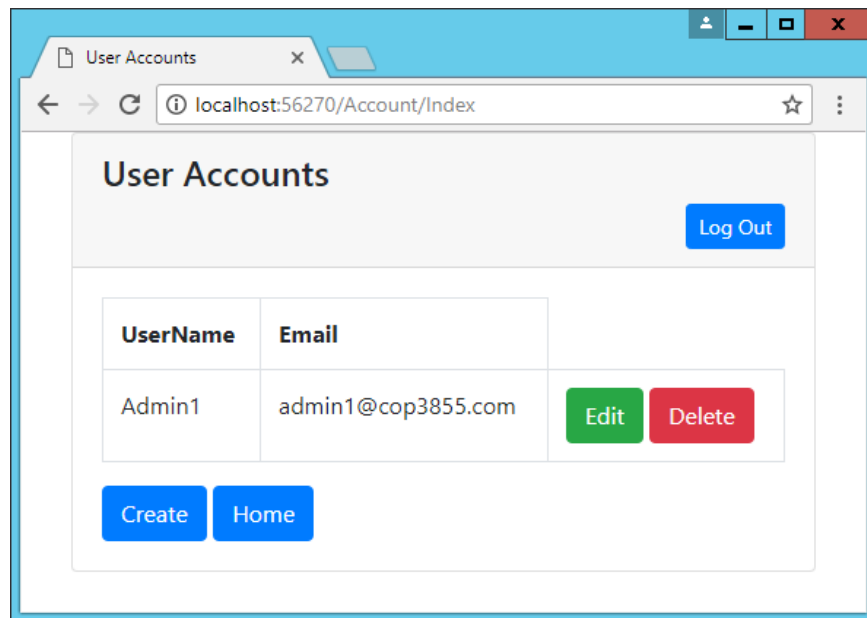


Figure 76. Listing of User Accounts

1.74 Task 68: Testing the Security Policy

Since you are presently unauthenticated and you can try to access features that requires authorization. Try accessing `/Admin/Index` URL, your browser will be redirected to the `/Account/Login` URL. Enter email address and password, and then submit the form. The Account controller will check the credentials you provided with users table—assuming you entered the right details—authenticate you and redirect you back to the `/Account/Login` URL, to which you now can have access. Figure 77 illustrates the process.

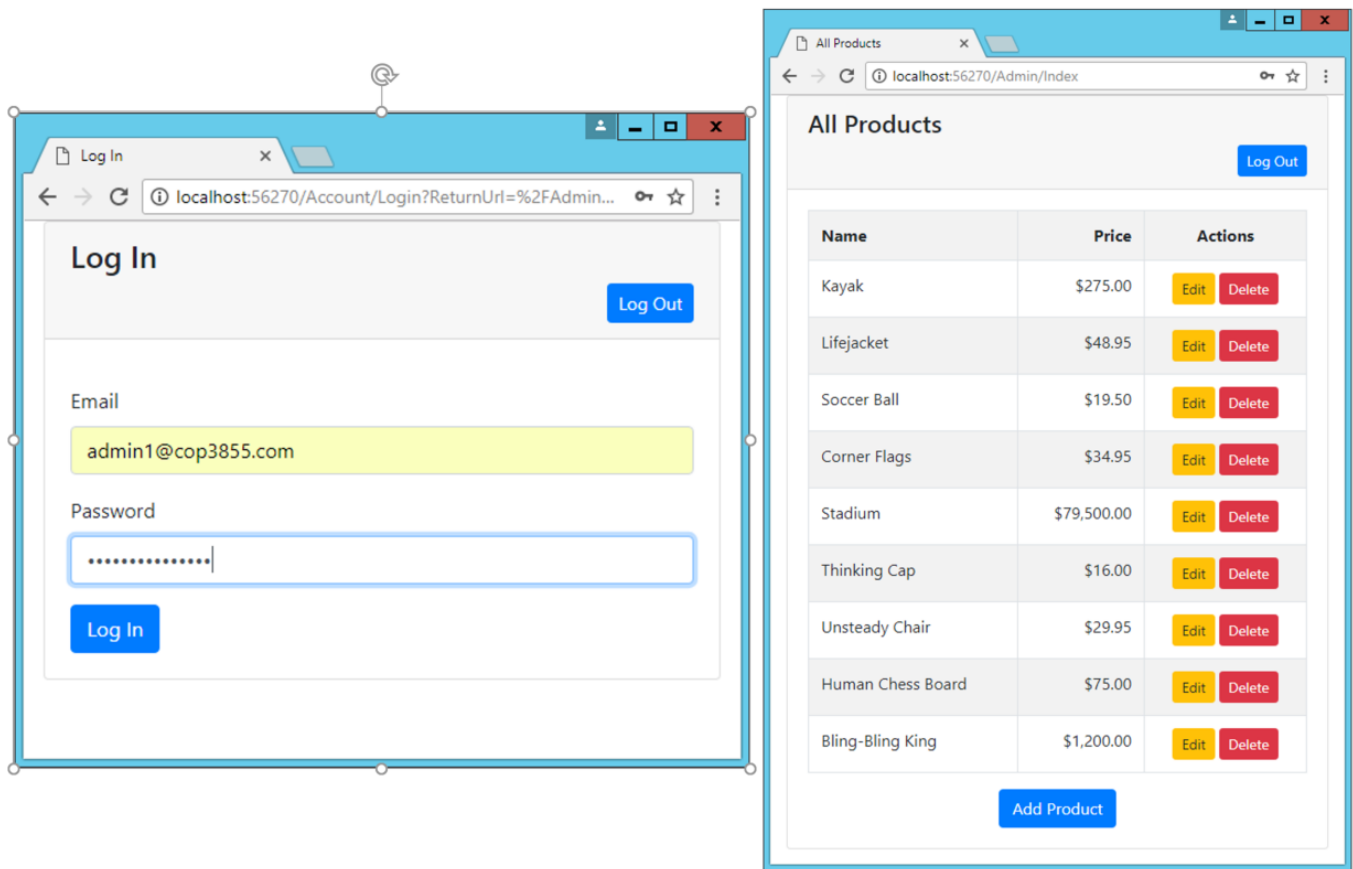


Figure 77. The administration authentication/authorization process

1.75 Summary

In conclusion, this tutorial demonstrates how the ASP.NET Core MVC can be used to create a realistic e-commerce application. This extended example introduced many key MVC features: controllers, action methods, routing, views, metadata, validation, layouts, authentication, and more. You also saw how some of the key technologies related to MVC can be used. These included the Entity Framework Core, and dependency injection. The result is an application that has a clean, component-oriented architecture that separates the various concerns and a code base that will be easy to extend and maintain. And that's the end of the SportsStore application.