ASP.NET Core Activity #3

Part 1 Display Data in a Table

Goal:

In this activity, you use .NET MVC Views to:

- Write Razor syntax on a webpage.
- Generate sample data.
- Create a data table in a SQL database.
- Query the database using SQL statements.
- Pass data to a Razor form and display the contents in a table.

What is Razor?

Razor is a Microsoft language used to generate HTML code on a .NET server. Similar technology exists in PHP and Java. A webpage contains a mix of Razor code and HTML code. The Razor code is processed by the server before being displayed to the browser. In the example to the right, each line that begins with the @ symbol is Razor code.

```
Index.cshtml   ProductsController.cs
@model IEnumerable<HelperSample.Models.Product>

<h2>Products</h2>

<ul>
    @foreach (var product in Model) {
        <li>
            <span class="producttitle">
              @product.Name
            </span>

            <span class="description">
              @product.Description
            </span>

            <span class="price">
              @if (product.UnitPrice == 0) {
                  <span>FREE!</span>
              }
              else{
                  @String.Format("{0:C2}", product.UnitPrice)
              }
            </span>
        </li>
    }
</ul>
```

Create a list of objects and display it in a table.

This activity is a continuation of the Activity #1.

1. Open the **Products Controller** and the Products **Index View** to review what we accomplished in previous tutorials.

The Products Controller managed a few methods as shown here.

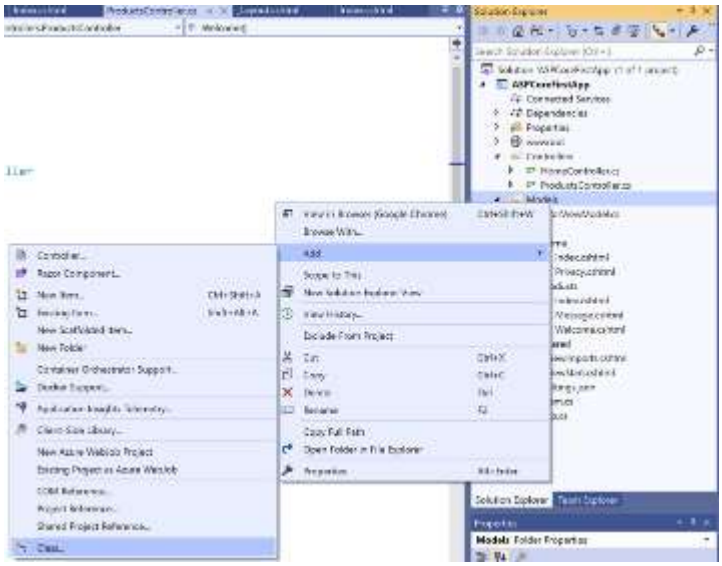
```
using Microsoft.AspNetCore.Mvc;

namespace ASPCoreFirstApp.Controllers
{
    public class ProductsController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome()
        {
            ViewBag.name = "Shad";
            ViewBag.secretNumber = 13;
            return View();
        }

    }
}
```

The Products **Index View** promised greater things to come. We will add a feature to the app where a list of products will be shown.

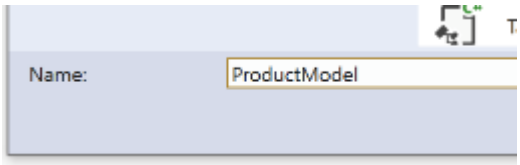
```
Welcome.cshtml | Message.cshtml | Program.cs | Startup.cs | Index.cshtml | ProductsController.cs | Layout.cshtml | Index.cshtml

@*
    For more information on enabling MVC for empty projects, visit http://go.microsoft.com/fwlink/?
      LinkID=397860
*@

<h1>Products Index Page</h1>
<p>We will display a list of products here very soon.</p>
```

Showing a Product List

In this section, we will create a list of products, store them in a database and display them on a webpage.



1. Create a Products Model class.
    a. **Right-click** in the Models folder, choose **Add > Class.**



    b. Name the class **ProductModel**.

2. Add some properties to define a product. Notice that I choose three different data types: **integer**, **string**, and **decimal**. Create a parameterized constructor method.

```
namespace ASPCoreFirstApp.Models
{
    public class ProductModel
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public string Description { get; set; }

        public ProductModel(int id, string name, decimal price, string description)
        {
            Id = id;
            Name = name;
            Price = price;
            Description = description;
        }
    }
}
```

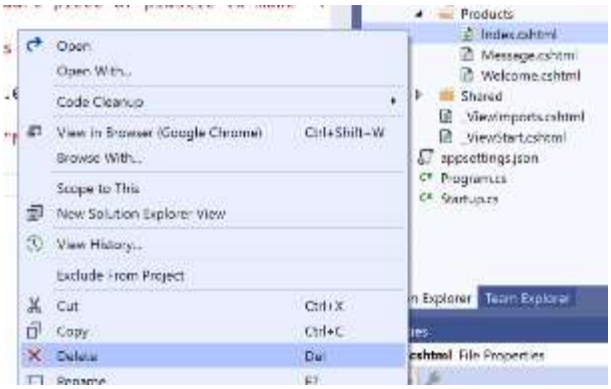
3. Add some hardcoded data into a **List of products** inside the Products controller. Pass the list of data to the View as shown here. Later, we will replace this hardcoded data with information from a database.

```
namespace ASPCoreFirstApp.Controllers
{
    public class ProductsController : Controller
    {
        public IActionResult Index()
        {
            List<ProductModel> productList = new List<ProductModel>();

            productList.Add(new ProductModel(1, "Mouse Pad",  5.99m, "A square piece of plastic to make
              mousing easier"));
            productList.Add(new ProductModel(2, "Web Cam", 45.50m, "Enables you to attend more Zoom
              meetings."));
            productList.Add(new ProductModel(3, "4 TB USB Hard Drive", 130.00m, "Back up all of your
              work.  You won't regret it."));
            productList.Add(new ProductModel(4, "Wireless Mouse", 15.99m, "Notebook mice really don't
              work very well. Do they?"));

            return View(productList);
        }
```

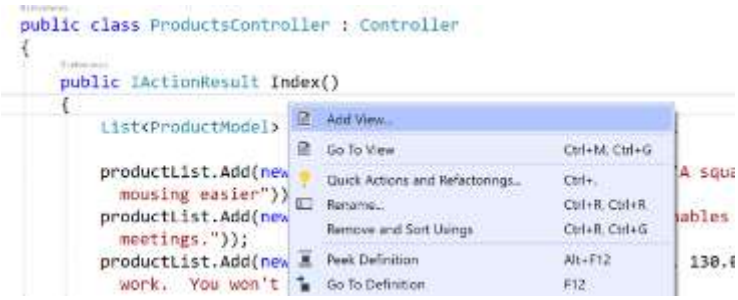
Invent some product instances and add them to the list.

The productList is passed to the index view as a parameter.

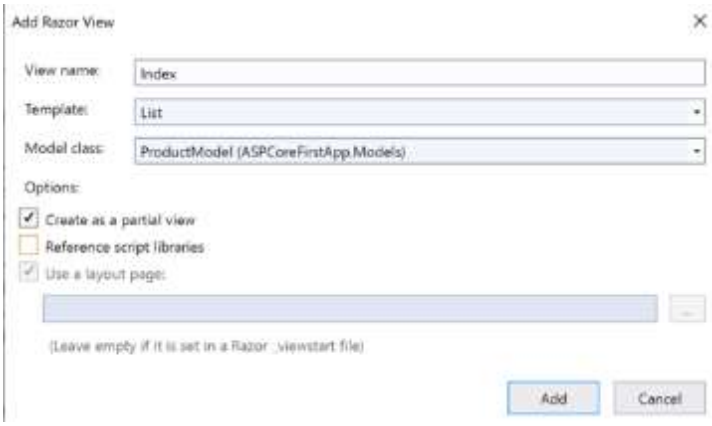
4. Delete the existing **Index** file inside the **Views/Products** folder. We will replace it with a new version shortly.



5. Right-click inside the **Index** method and choose **Add View**…



6. Use the following configuration to create the view: **name** = index, **Template** = list, **Model** = product.



The generated code should show a <table> element with a table head <thead>, table header elements <th>, table rows <tr>, and table data elements <td>.

Let's highlight some important parts of the page.

Index.cshtml | ProductModel.cs | Welcome.cshtml | Message.cshtml | Program.cs | Startup.cs

```
1   @model IEnumerable<ASPCoreFirstApp.Models.ProductModel>
2
3   <p>
4       <a asp-action="Create">Create New</a>
5   </p>
6   <table class="table">
7       <thead>
8           <tr>
9               <th>
10                  @Html.DisplayNameFor(model => model.Id)
11              </th>
12              <th>
13                  @Html.DisplayNameFor(model => model.Name)
14              </th>
15              <th>
16                  @Html.DisplayNameFor(model => model.Price)
17              </th>
18              <th>
19                  @Html.DisplayNameFor(model => model.Description)
20              </th>
21              <th></th>
22          </tr>
23      </thead>
24      <tbody>
25  @foreach (var item in Model) {
26          <tr>
27              <td>
28                  @Html.DisplayFor(modelItem => item.Id)
29              </td>
30              <td>
31                  @Html.DisplayFor(modelItem => item.Name)
32              </td>
33              <td>
34                  @Html.DisplayFor(modelItem => item.Price)
35              </td>
36              <td>
37                  @Html.DisplayFor(modelItem => item.Description)
38              </td>
```

@model

This tells the page what data type was passed to it from the controller. IEnumerable is an interface for the list<> data type.

@Html

These html "helper" elements dynamically generate html tags in the rendered page.

=>

Arrow functions are not required, but help with how an item is formatted. See explanation below.

7. Run the application and **inspect** the HTML elements in the browser. The following image shows the developer tools inspecting the page in Chrome. You can see that the **@Html helper tags** generated **standard HTML code** in the browser.
8. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

The arrow functions, officially called **Lambda Expressions**, used in the Razor pages seem completely unnecessary at first. The following Razor code would work equally well as the code generated for us.

Version 1: The generated code for the table	Version 2: Simpler code that you could alternatively use. This does not utilize lambda functions.

`@foreach (var item in Model) {`
`<tr>`
`<td>`
`@Html.DisplayFor(modelItem => item.Id)`
`</td>`
`<td>`
`@Html.DisplayFor(modelItem => item.Name)`
`</td>`
`<td>`
`@Html.DisplayFor(modelItem => item.Price)`
`</td>`
`<td>`
`@Html.DisplayFor(modelItem => item.Description)`
`</td>`
`<td>`
`@Html.ActionLink("Edit", "Edit", new { /* id=ite`
`@Html.ActionLink("Details", "Details", new { /*`
`@Html.ActionLink("Delete", "Delete", new { /* i`
`</td>`
`</tr>`
	`<tbody>`
	`@foreach (var item in Model) {`
	`<tr>`
	`<td>`
	`@item.Id`
	`</td>`
	`<td>`
	`@item.Name`
	`</td>`
	`<td>`
	`@item.Price`
	`</td>`
	`<td>`
	`@item.Description`
	`</td>`
	`<td>`
	`@Html.ActionLink("Edit",`
	`@Html.ActionLink("Detail`
	`@Html.ActionLink("Delete`
	`</td>`
	`</tr>`

The Purpose of the Lambda Expressions

Lambda expressions (a.k.a. arrow functions) help with formatting data. ASP.NET allows you to modify the display properties and format of each property inside a class. For example, try these modifications shown below in the **ProductModel** class. The **display name** is changed for each item and the **currency** format is applied to the price property. You will have to do some imports to use the notations.

```
public class ProductModel
{
    [DisplayName("Id number")]
    public int Id { get; set; }
    [DisplayName("Product Name")]
    public string Name { get; set; }

    [DataType(DataType.Currency)]
    [DisplayName("Cost to customer")]
    public decimal Price { get; set; }
    [DisplayName("What you get...")]
    public string Description { get; set; }
```

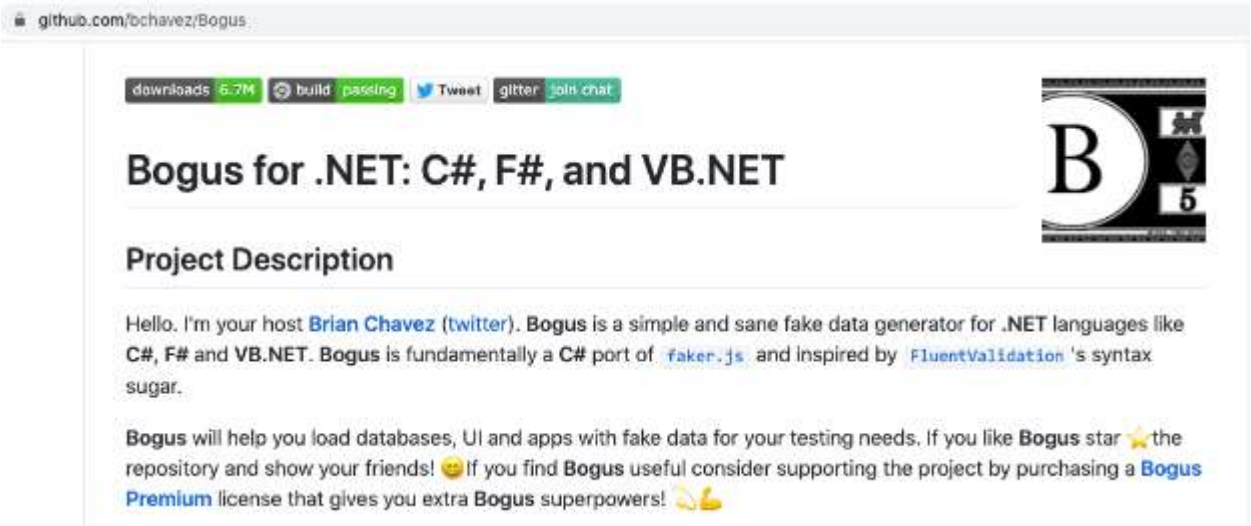
The resulting changes appear in the following picture. Notice the **column header names** have changed and the **price has a dollar sign**. The arrow functions in the View code make this formatting possible.

9. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Id number	Product Name	Cost to customer	What you get...
1	Mouse Pad	$5.99	A square piece of plastic to make mousing easier
2	Web Cam	$45.50	Enables you to attend more Zoom meetings.
3	4 TB USB Hard Drive	$130.00	Back up all of your work. You won't regret it.
4	Wireless Mouse	$15.99	Notebook mice really don't work very well. Do they?
5	Practical Cotton Table	$43.87	I tried to grab it but got bonbon all over it.

Generate Fake Data

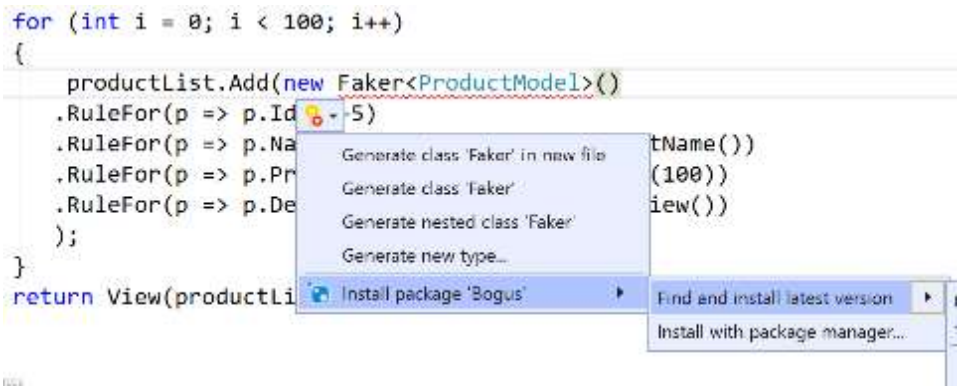
For an easy and humorous way to generate fake data, we are going to use a library called **Faker** or **Bogus**. Here is the GitHub resource to see the complete documentation.



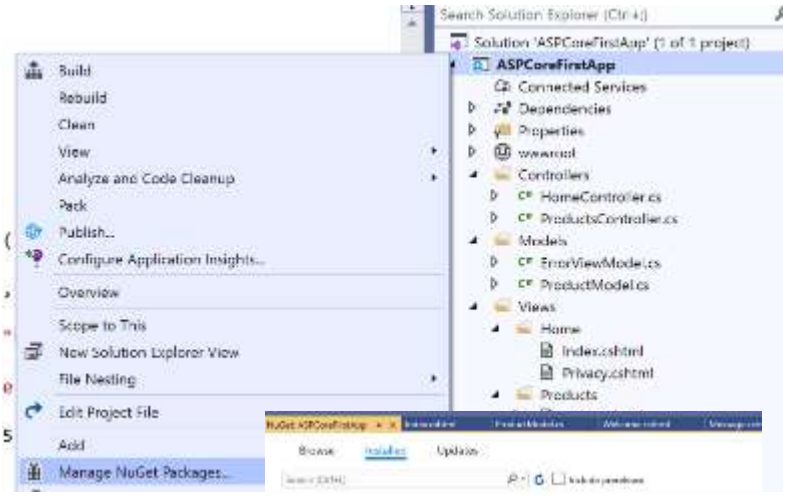
1. Modify the Products controller to the following code to **include 100 more products**.

```
public class ProductsController : Controller
{
    public IActionResult Index()
    {
        List<ProductModel> productList = new List<ProductModel>();

        productList.Add(new ProductModel(1, "Mouse Pad",  5.99m, "A square piece of plastic to
          make mousing easier"));
        productList.Add(new ProductModel(2, "Web Cam", 45.50m, "Enables you to attend more Zoom
          meetings."));
        productList.Add(new ProductModel(3, "4 TB USB Hard Drive", 130.00m, "Back up all of your
          work.  You won't regret it."));
        productList.Add(new ProductModel(4, "Wireless Mouse", 15.99m, "Notebook mice really don't
          work very well. Do they?"));
        for (int i = 0; i < 100; i++)
        {
            productList.Add(new Faker<ProductModel>()
           .RuleFor(p => p.Id, i+5)
           .RuleFor(p => p.Name, f => f.Commerce.ProductName())
           .RuleFor(p => p.Price, f => f.Random.Decimal(100))
           .RuleFor(p => p.Description, f => f.Rant.Review())
           );
        }
        return View(productList);
    }
```

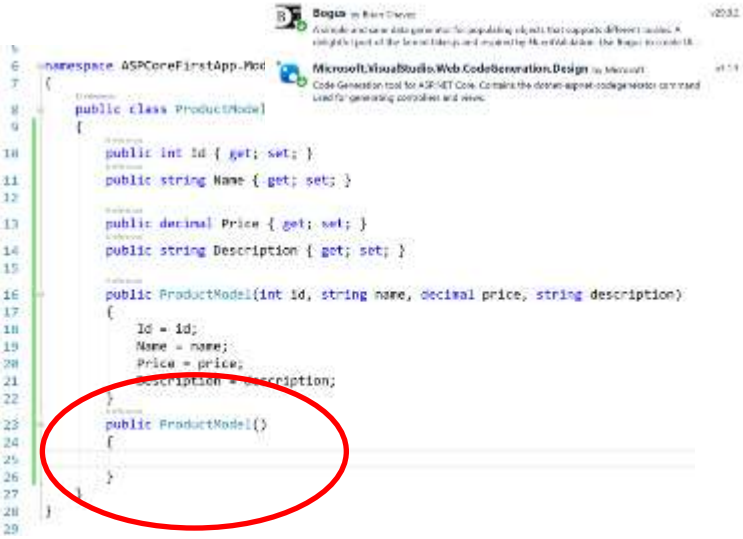
Faker< > is a function from the **Bogus** library, which is a dependency available via the **Nuget** package manager. You should be able to install Bogus the moment you type Faker<ProductsModel>. Visual Studio intelli-sense will recognize the need for a dependency.



Once Bogus is added to the project, you can check on its status by right-clicking the solution title and choosing Manage NuGet Packages.



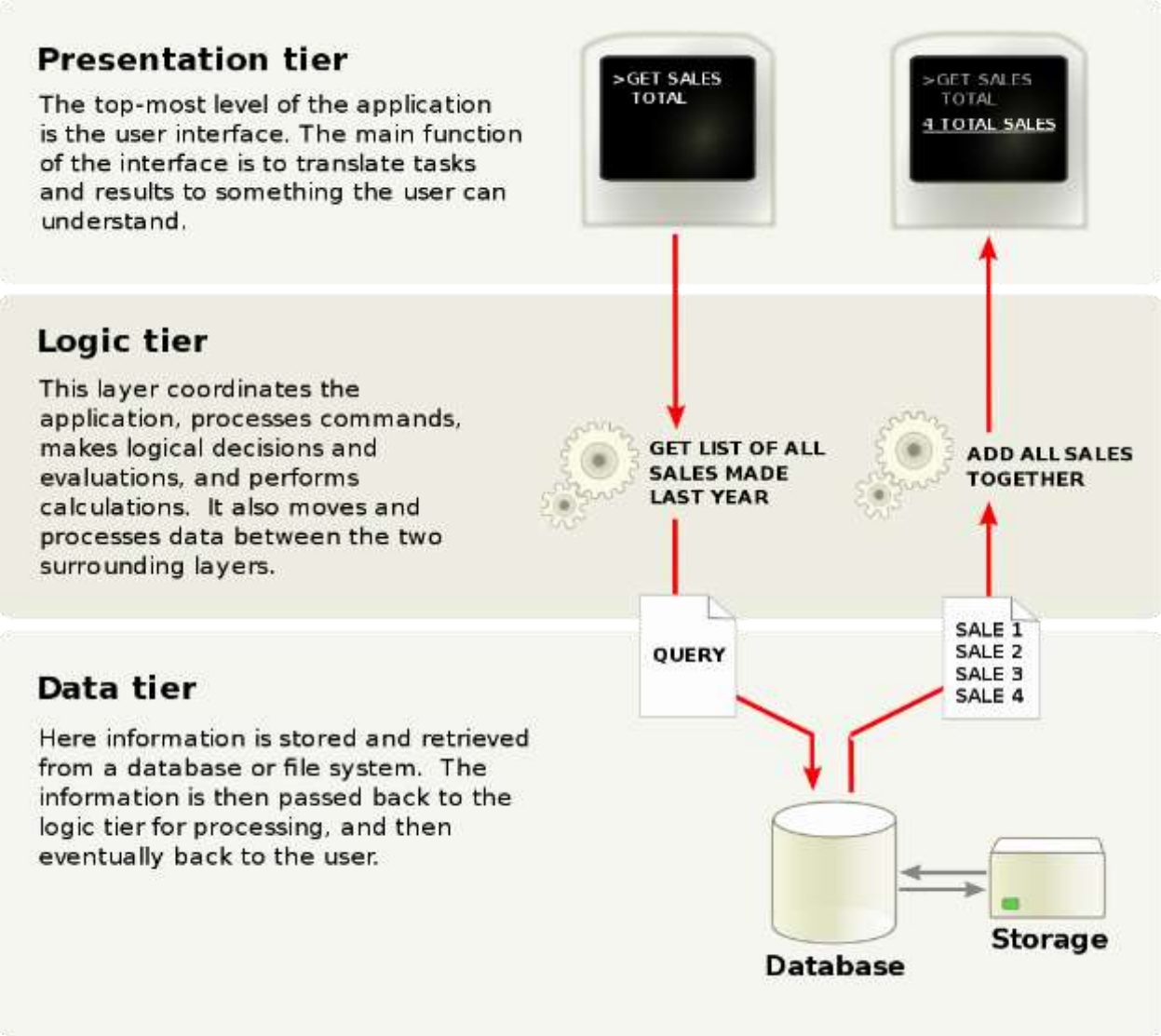
Bogus requires that the ProductModel class have an **empty constructor**, shown here.



Data Repository

Next, we are going to refactor the application to reflect a better design pattern. We are going to take the sample data that is defined in the Products Controller and move it to a data service layer.

Consider the following diagram of a **3-tier application**. Applications that scale to large user numbers well are typically separated into layers according to specific tasks. Splitting the work allows multiple servers to handle the workload.



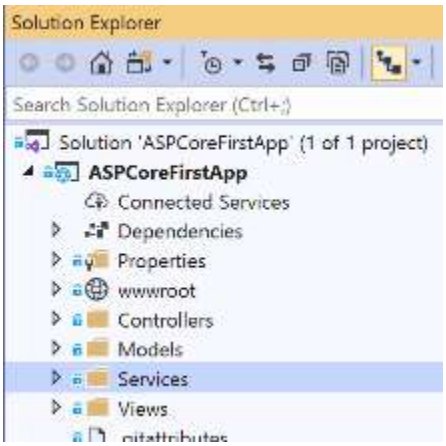
In our application:

- the View components (HTML + Razor) correspond to the **Presentation Layer**.
- the controller corresponds to the **Logic Tier**.
- the Data Tier is **currently absent** from our application.

We currently have stored the **list of products** in the controller. We will now create the **Data Tier** layer and provide a pathway to making the data storage a distinct module of the application that can be easily changed according to our storage needs. We will eventually replace the mock data source we currently have with a **SQL database**.

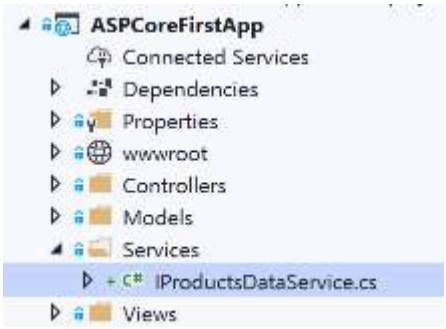
Create an Interface to Define the Data Service

1. Create a new folder called **Services** in the project.



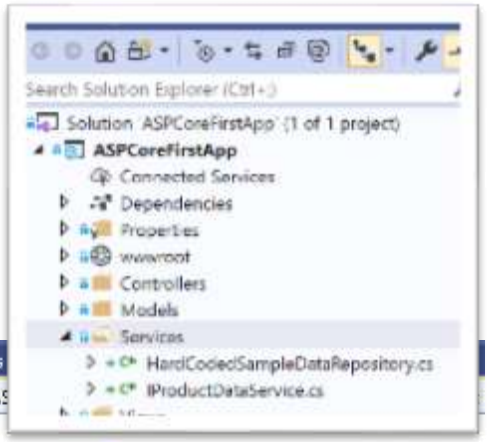
2. Create a new class called **IProductsDataService** in the Services folder.

3. Add the following code to the new class.



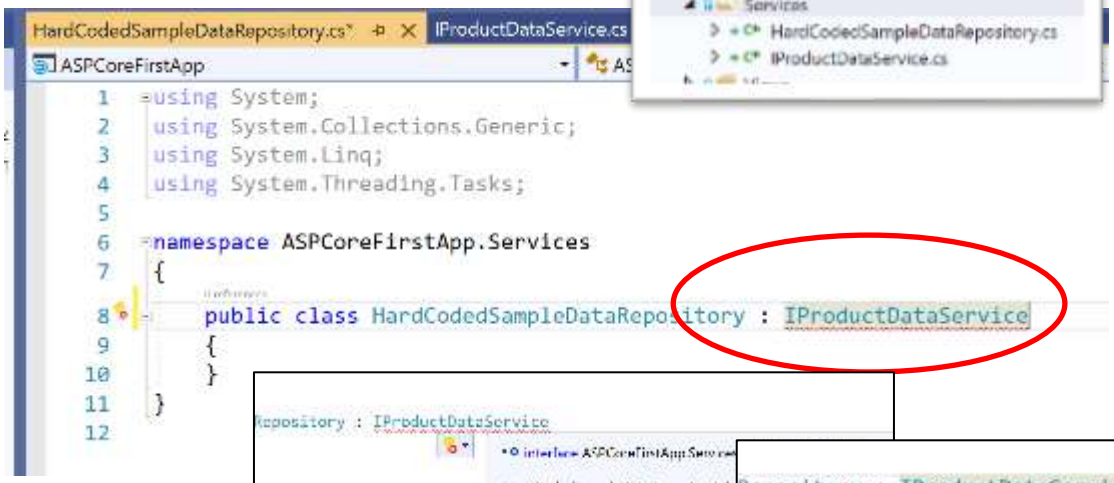
This interface will define three operations that our application will eventually be able to perform. These are the basic **CRUD** (Create Read Update Delete) operations that are common in data storage applications.

4. Create another class in the Services folder. Name it **HardCodedSampleDataRepository**.cs



5. Add the implementation of the new interface we just defined.

Notice that the red-line error on the new code says we need to implement the methods defined in the interface.



6. Choose **Implement interface** from the "Show potential fixes" menu.

7. All of the methods defined in the interface should now appear in the new service.

8. Cut all of the Fake data generation code from the **ProductsController** file.

```
13  public class ProductsController : Controller
14  {
15      public IActionResult Index()
16      {
17
18          List<ProductModel> productList = new List<ProductModel>();
19
20          productList.Add(new ProductModel(1, "Mouse Pad", 5.99m, "A square piece of plastic to make mousing
              easier"));
21          productList.Add(new ProductModel(2, "Web Cam", 45.50m, "Enables you to attend more Zoom meetings"));
22          productList.Add(new ProductModel(3, "4 TB USB Hard Drive", 130.00m, "Back up all of your work. You
              won't regret it."));
23          productList.Add(new ProductModel(4, "Wireless Mouse", 15.99m, "Notebook mice really don't work very
              well. Do they?"));
24          for (int i = 0; i < 100; i++)
25          {
26              productList.Add(new Faker<ProductModel>()
27               .RuleFor(p => p.Id, i + 5)
28               .RuleFor(p => p.Name, f => f.Commerce.ProductName())
29               .RuleFor(p => p.Price, f => f.Random.Decimal(100))
30               .RuleFor(p => p.Description, f => f.Rant.Review())
31              );
32          }
33
34          return View(productList);
35      }
```

9. In the **HardCodedSampleDataRepository** file, create a constructor and paste the code inside. Make modifications as shown here.

```
namespace ASPCoreFirstApp.Services
{
    public class HardCodedSampleDataRepository : IProductDataService
    {
        // use static to ensure the data set does not change
        static List<ProductModel> productList;

        // create the list in the constructor of the service.
        public HardCodedSampleDataRepository()
        {
            productList = new List<ProductModel>();
            productList.Add(new ProductModel(1, "Mouse Pad", 5.99m, "A square piece of plastic to make mousing
              easier"));
            productList.Add(new ProductModel(2, "Web Cam", 45.50m, "Enables you to attend more Zoom meetings"));
            productList.Add(new ProductModel(3, "4 TB USB Hard Drive", 130.00m, "Back up all of your work. You
              won't regret it."));
            productList.Add(new ProductModel(4, "Wireless Mouse", 15.99m, "Notebook mice really don't work very
              well. Do they?"));
            for (int i = 0; i < 100; i++)
            {
                productList.Add(new Faker<ProductModel>()
                    .RuleFor(p => p.Id, i + 5)
                    .RuleFor(p => p.Name, f => f.Commerce.ProductName())
                    .RuleFor(p => p.Price, f => f.Random.Decimal(100))
                    .RuleFor(p => p.Description, f => f.Rant.Review())
                );
            }
        }
        public List<ProductModel> AllProducts()
        {
            return productList;
        }

```

productList is a class-member variable, accessible throughout all of the methods below.

The data is randomly generated inside the constructor.

The first method only needs to return the entire productList.

10. In the ProductsController, make the following changes.

```
namespace ASPCoreFirstApp.Controllers
{
    public class ProductsController : Controller
    {
        public IActionResult Index()
        {
            HardCodedSampleDataRepository repository = new HardCodedSampleDataRepository();
            return View(repository.AllProducts());
        }
```

Instantiate the data repository. This repository will be replaced in a future version of the application using a technique called Dependency Injection.

Get all of the data records via the AllProduct method.

11. Run the program. There should be no visual or behavior changes to the application. We have simply refactored the application to implement a better code structure.

Coding Challenge: Create another model
To reinforce the process of managing models, controller, and views, create another model class in addition to the Products Model. This will go in the Models folder. Create a controller and View to display the new item in a table.

1. Create a class for another object which represents a product (**Movie, Car, Airline Ticket**, etc.) with at least the following properties: String, Integer, and Date.
2. Create another **Controller** and **View** based on the example created in this exercise.
3. Generate some **sample data** for the new item.
4. Display the resulting list in a **table**.
5. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

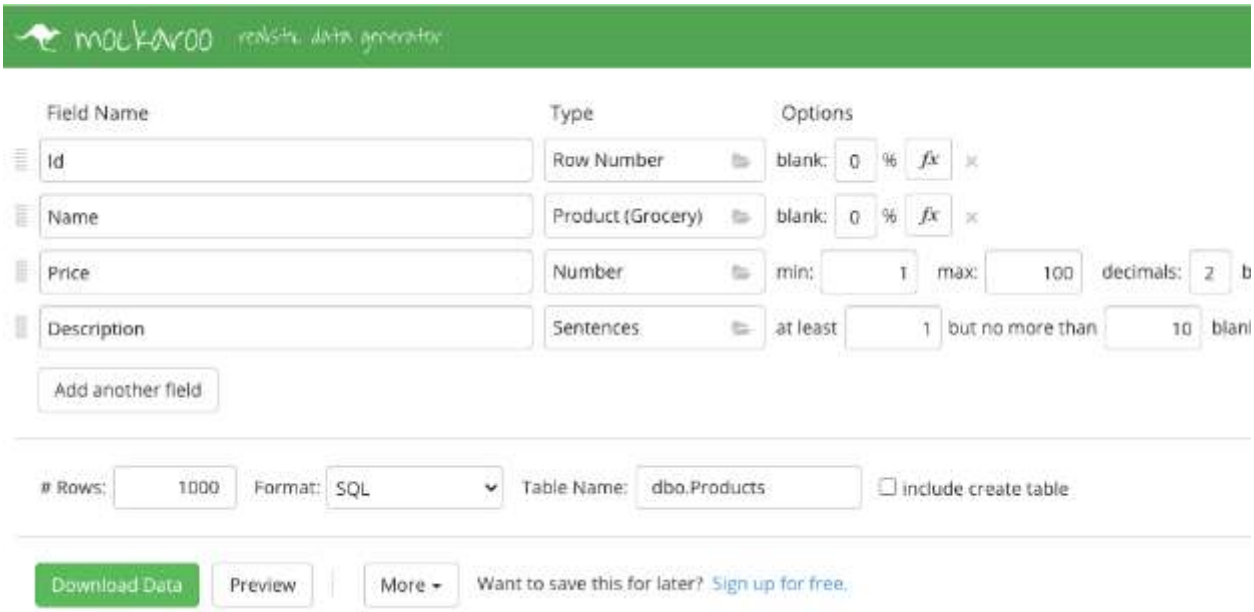
Database table for products

In this section, we will create a database source for product information replacing the process of generating the data within the controller from the Bogus library.

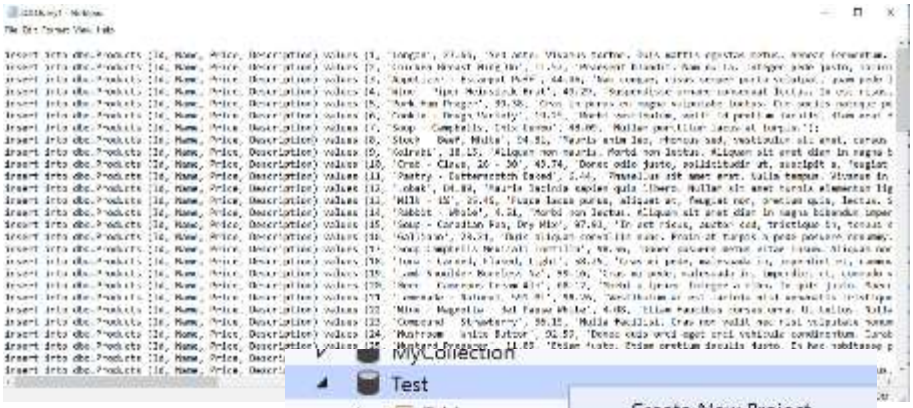
1. In the test database (where the users table already exists), create a new table called **dbo.Products** using the values shown here.

2. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

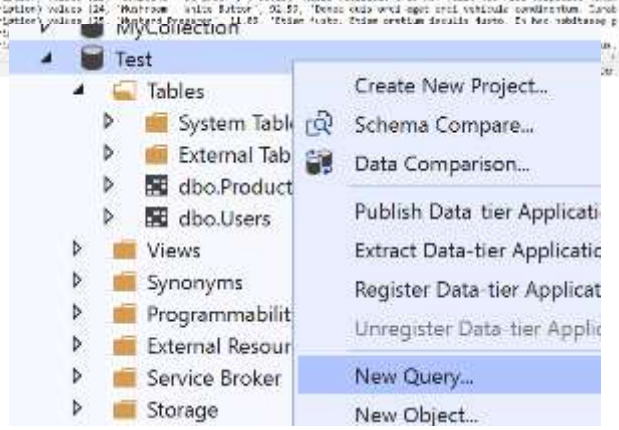
3. Fill the table with some sample data. Use the site Mockaroo to generate 1000 products automatically. Import the SQL file into the database. Here is an example of one way you could generate sample data. Take a screenshot of the database at this stage. Paste it into a Word document and caption the image with a brief explanation of what you just demonstrated.



4. Download the data and open it in a text editor.

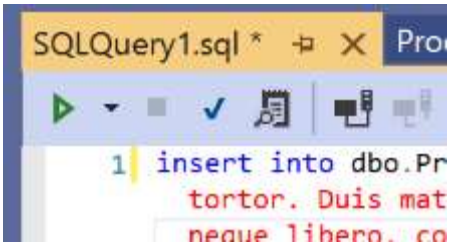


5. In the SQL Server window, right-click on the test database and start a **New Query**…



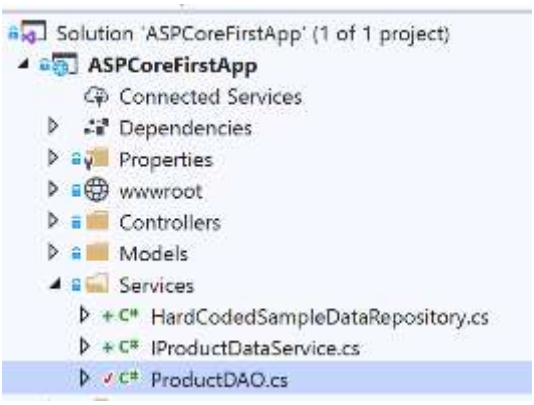
6. **Paste** the SQL statements from Mockaroo into the query window.

7. **Execute** the query by clicking the **green triangle** at the top of the query window.



You should see the database filled with values.

8. In the **Services folder** in the project, add a **ProductsDAO** class.



9. Implement the **IProductDataServices** interface.

```
using ASPCoreFirstApp.Models;
using System;
using System.Collections.Generic;
using System.Data.SqlClient;

namespace ASPCoreFirstApp.Services

{
    public class ProductDAO : IProductDataService
    {

    }
}
```

```
namespace ASPCoreFirstApp.Services

{
    public class ProductDAO : IProductDataService
    {

    }
}
```

Implement interface
Implement all members explicitly

10. Implement all the methods.

11. Add code to the method **AllProducts** that connects to the database and returns all records from the Products table.

```
public class ProductDAO : IProductDataService
{
    // copy this string from the test database properties values.
    string connectionString = @"Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Test;Integrated Security=True;Connect
      Timeout=30;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False";

    public List<ProductModel> AllProducts()
    {
        // assume nothing is found
        List<ProductModel> foundProducts = new List<ProductModel>();

        // uses prepared statements for security. @username @password are defined below
        string sqlStatement = "SELECT * FROM dbo.Products";

        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            SqlCommand command = new SqlCommand(sqlStatement, connection);


            try
            {
                connection.Open();
                SqlDataReader reader = command.ExecuteReader();

                while (reader.Read())
                {
                    foundProducts.Add(new ProductModel((int)reader[0], (string)reader[1], (decimal)reader[2], (string)
                      reader[3]));

                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            };
        }
        return foundProducts;

    }
```

12. Add another method to perform a search by name.

```
public List<ProductModel> SearchProducts(string searchTerm)
{
    // assume nothing is found
    List<ProductModel> foundProducts = new List<ProductModel>();

    // uses prepared statements for security. @username @password are defined below
    string sqlStatement = "SELECT * FROM dbo.Products WHERE Name LIKE @Name";

    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        SqlCommand command = new SqlCommand(sqlStatement, connection);

        // define the values of the two placeholders in the sqlStatement string
        command.Parameters.AddWithValue("@Name", '%' + searchTerm + '%');


        try
        {
            connection.Open();
            SqlDataReader reader = command.ExecuteReader();

            while (reader.Read())
            {
                foundProducts.Add(new ProductModel((int)reader[0], (string)reader[1], (decimal)reader[2], (string)
                  reader[3]));

            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        };
    }
    return foundProducts;
}
```

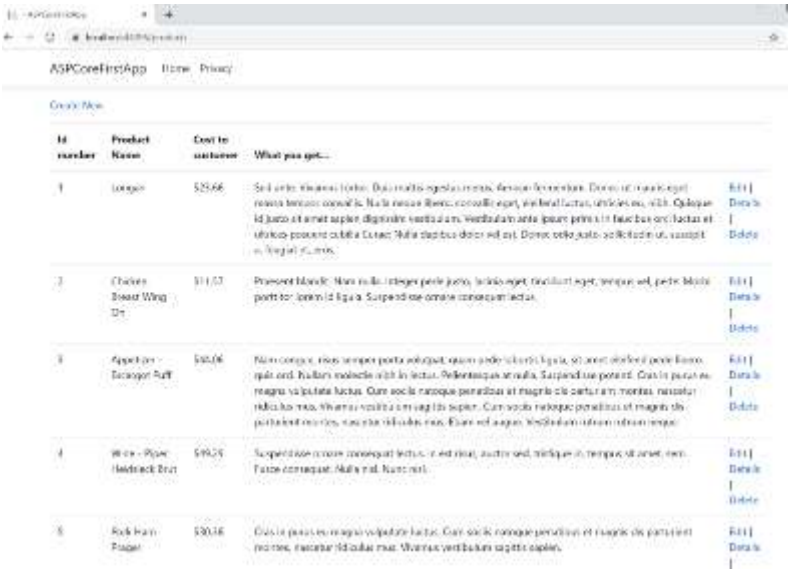
Microsoft has examples for all types of SQL queries at their official documentation
https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/retrieving-and-modifying-data

Other methods, such as **delete**, **selectOne**, and **update** are future enhancements for the application and are not required at this time.

13. In the **ProductsController**, use the **ProductsDAO** to **fetch the data** from the database instead of generating new fake data in the controller.

```
namespace ASPCoreFirstApp.Controllers
{
    public class ProductsController : Controller
    {
        ProductDAO repository = new ProductDAO();
        public ProductsController()
        {
            repository = new ProductDAO();
        }

        public IActionResult Index()
        {
            return View(repository.AllProducts());
        }
```

14. The app should display the contents of the database instead of the Bogus data.
15. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.



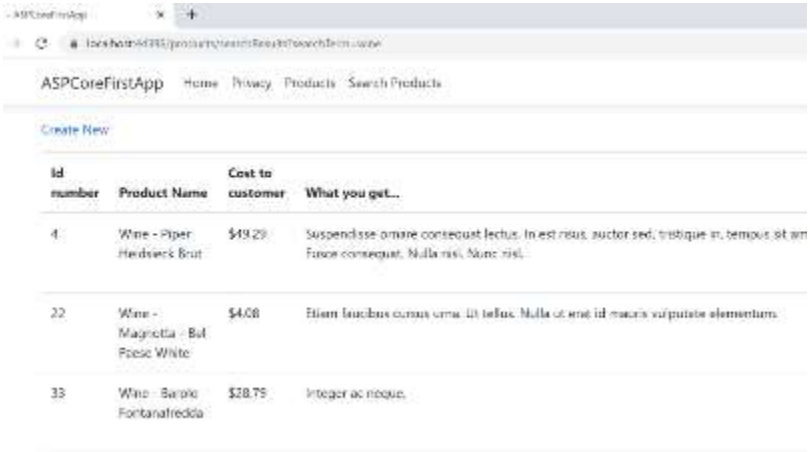
16. Add **another method** to the **ProductsController** to respond to search results.

```
public class ProductsController : Controller
{
    ProductDAO repository = new ProductDAO();
    public ProductsController()
    {
        repository = new ProductDAO();
    }

    public IActionResult Index()
    {
        return View(repository.AllProducts());
    }

    public IActionResult SearchResults(string searchTerm)
    {
        List<ProductModel> productList = repository.SearchProducts(searchTerm);
        return View("Index", productList);
    }
```

17. You should be able to perform a search with the following URL:

https://localhost:44396/products/searchResults?searchTerm=wine

18. Take a **screenshot** of the app at this stage. Paste it into a Word document and caption the image with a brief explanation of what you just demonstrated.

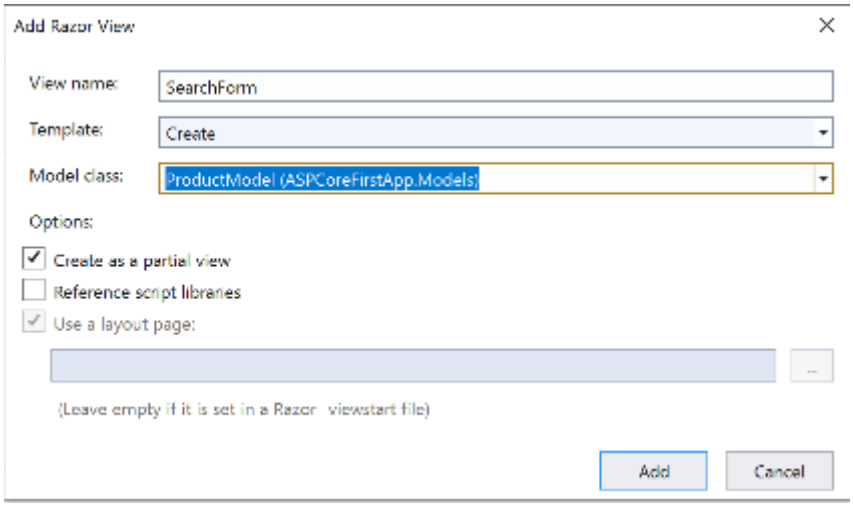


Let’s add a form that will allow the user to perform a search.

19. Add a new method in the **ProductsController** called **SearchForm**.

```
21      public IActionResult Index()
22      {
23          return View(repository.AllProducts());
24      }
25
26      public IActionResult SearchResults(string searchTerm)
27      {
28          List<ProductModel> productList = repository.SearchProducts(searchTerm);
29          return View("Index", productList);
30      }
31
32      public IActionResult SearchForm()
33      {
34          return View();
35      }
36
```

20. Right-click inside the SearchForm method and choose Add View…
21. Choose Razor View.
22. Choose name = SearchForm, Template = Create, Model class = ProductModel.



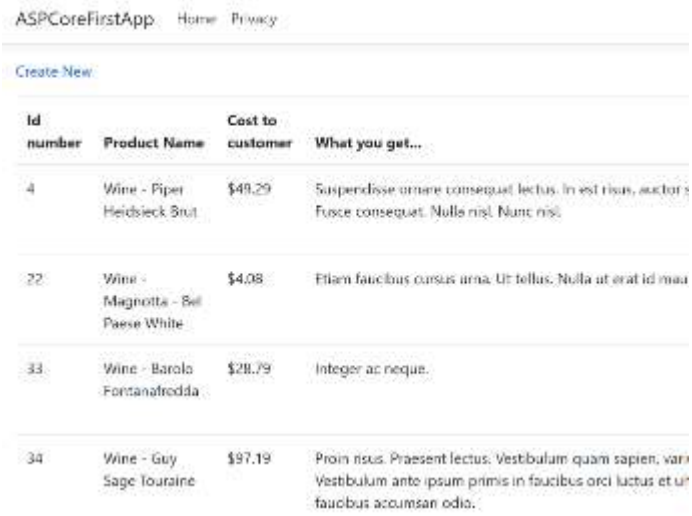
23. Modify the resulting HTML code for a data entry form.
24. Run the application and use the URL:

https://localhost:44396/products/searchForm

25. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

26. The new form should produce search results according to the keyword you supply.

27. Take a screenshot of the app at this stage. Paste it into a Word document and caption the image with a brief explanation of what you just demonstrated.



Add Links in the Navbar

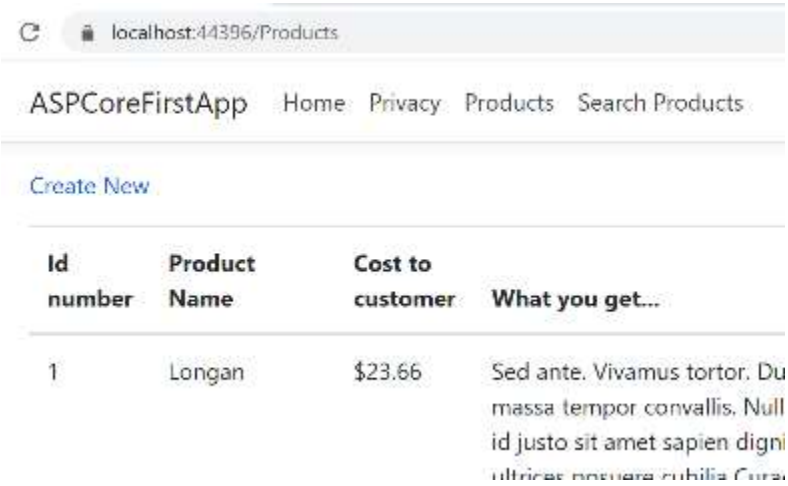
To make this app much more usable, add some links to the Navbar to allow the user to open the Products page as well as the SearchForm page.

1. Open **_Layout.cshtml** in Views > Shared folder.
2. Add two more links in the navbar section as shown here. Copy and paste a previous link and modify the details.

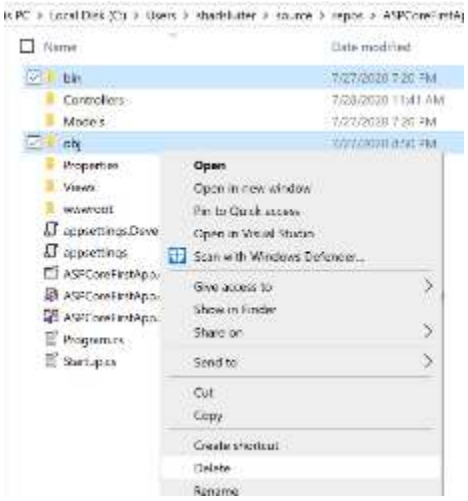
```
10  <body>
11      <header>
12          <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
13              <div class="container">
14                  <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">ASPCoreFirstApp</a>
15                  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-collapse" aria-controls="navbarSupportedContent"
16                          aria-expanded="false" aria-label="Toggle navigation">
17                      <span class="navbar-toggler-icon"></span>
18                  </button>
19                  <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
20                      <ul class="navbar-nav flex-grow-1">
21                          <li class="nav-item">
22                              <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
23                          </li>
24                          <li class="nav-item">
25                              <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
26                          </li>
27
28                          <li class="nav-item">
29                              <a class="nav-link text-dark" asp-area="" asp-controller="Products" asp-action="Index">Products</a>
30                          </li>
31
32                          <li class="nav-item">
33                              <a class="nav-link text-dark" asp-area="" asp-controller="Products" asp-action="SearchForm">Search Products</a>
34                          </li>
35
36
37                      </ul>
38                  </div>
```

You should be able to navigate to the two new pages.

3. Take a screenshot of the app at this stage. Paste it into a Word document and caption the image with a brief explanation of what you just demonstrated.



Deliverables:



1. This activity has multiple parts. Complete all parts before submitting.
2. Submit a Microsoft Word document with screenshots of the application being run at each stage of development. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
4. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.

Part 2 Working with Bootstrap

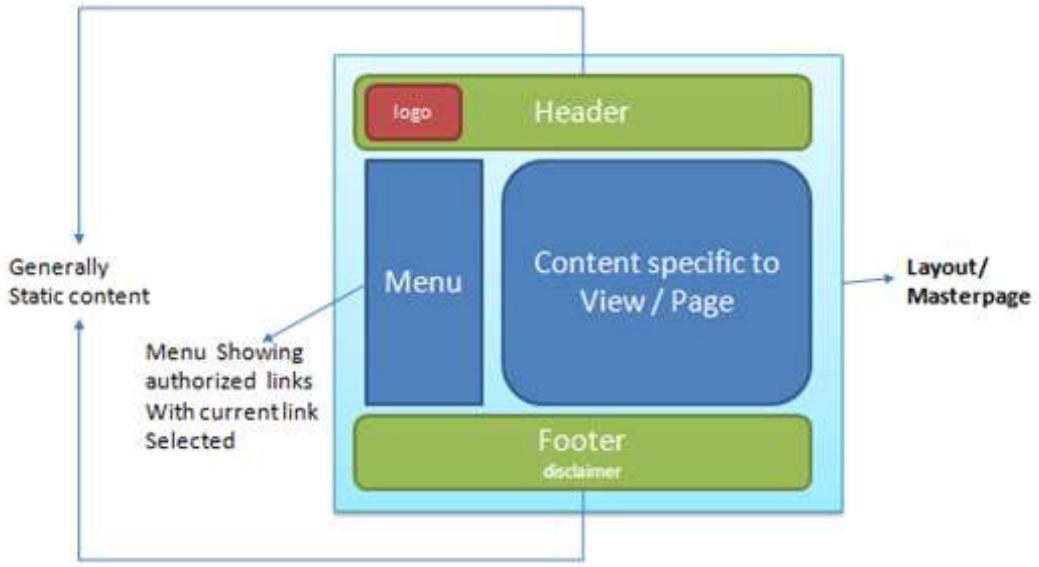
Goal:

In this activity, you will:

- Use Razor Layouts.
- Format the site using Bootstrap CSS.
- Change the default Bootstrap CSS theme.
- Use Flexbox as an alternative to a table.

About Layouts

Layouts are designed to reduce code duplication. We create a piece of the page, such as the header or footer, only once and reference it in a layout file. Consider the part of a website as shown here.



To use layouts in this example, we would design each region as a separate piece of html code and then combine them using one master layout file, pictured here as the light blue box surrounding all of the other pieces. The pieces are reassembled dynamically when the server generates a new webpage for a browser request.

1. Open the **_Layout.cshtml** file in the View > Shared folder.

Notice the sections where the pieces of a layout are assembled. The **@** symbol is used when in the dynamically-inserted sections.

```
31
32                      <li class="nav-item">
33                          <a class="nav-link text-dark" asp-area="" asp-controller="Products" asp-action="SearchForm">Search Products</a>
34                      </li>
35
36
37                  </ul>
38              </div>
39          </div>
40      </nav>
41  </header>
42  <div class="container">
43      <main role="main" class="pb-3">
44          @RenderBody()
45      </main>
46  </div>
47
48  <footer class="border-top footer text-muted">
49      <div class="container">
50          &copy; 2020 - ASPCoreFirstApp - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
51      </div>
52  </footer>
53  <script src="~/lib/jquery/dist/jquery.min.js"></script>
54  <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
55  <script src="~/js/site.js" asp-append-version="true"></script>
56  @RenderSection("Scripts", required: false)
57 </body>
58 </html>
```

@RenderBody() is a Razor function that inserts a View

Let's create a partial view for the navbar in order to make the _layout page smaller.

2. **Cut out the entire section** from <nav> to </nav> and put it into a separate file. Save the file as **_navbar.cshtml** in the shared folder. Here is the contents of the new file taken from layout.

navbar.cshtml | site.css | ProductDAO.cs | ProductModel.cs | ProductsController.cs | Index.cshtml | Welcome.cshtml | Layout.cs

```
1  <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
2      <div class="container">
3          <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">ASPCoreFirstApp</a>
4          <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-collapse" aria-controls="navbarSupportedContent"
5                  aria-expanded="false" aria-label="Toggle navigation">
6              <span class="navbar-toggler-icon"></span>
7          </button>
8          <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
9              <ul class="navbar-nav flex-grow-1">
10                 <li class="nav-item">
11                     <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Ho
12                 </li>
13                 <li class="nav-item">
14                     <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">P
15                 </li>
16
17                 <li class="nav-item">
18                     <a class="nav-link text-dark" asp-area="" asp-controller="Products" asp-action="Index
19                 </li>
20
21                 <li class="nav-item">
22                     <a class="nav-link text-dark" asp-area="" asp-controller="Products" asp-action="Search
23                 </li>
24
25
26             </ul>
27         </div>
28     </div>
29 </nav>
30
```

- Views
  - Home
    - Index.cshtml
    - Privacy.cshtml
  - Products
    - Index.cshtml
    - Message.cshtml
    - SearchForm.cshtml
    - Welcome.cshtml
  - Shared
    - Layout.cshtml
    - _navbar.cshtml
    - _ValidationScriptsPartial.cshtml
    - Error.cshtml
  - _ViewImports.cshtml
  - _ViewStart.cshtml

3. In the **_Layout.cshtml** file, replace the missing content with @Html.Partial("_navbar.cshtml"); as shown here.

```
9   </head>
10  <body>
11      <header>
12          @Html.Partial("_navbar.cshtml");
13      </header>
14      <div class="container">
15          <main role="main" class="pb-3">
16              @RenderBody()
17          </main>
18      </div>
19
20      <footer class="border-top footer text-muted">
21          <div class="container">
22              &copy; 2020 - ASPCoreFirstApp - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
23          </div>
24      </footer>
25      <script src="~/lib/jquery/dist/jquery.min.js"></script>
26      <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
27      <script src="~/js/site.js" asp-append-version="true"></script>
28      @RenderSection("Scripts", required: false)
29  </body>
30  </html>
31
```

4. Run the application to ensure it still shows the **nav bar** correctly. There should be **no visible change** to the application.

Visual Confirmation of Page Sections

Next, we will add some **CSS formatting** to visually **emphasize** the separate sections on the layout page.

1. Open the site.css in the folder **wwwroot** > **css**. Notice that line 27 of the layout file (pictured above) has a <script> reference to this file. The site.css file is the preferred location to customize the site.
2. To visually demonstrate the use of the template, we are going to add a **large border** around the <**main**> and <**header**> sections of the page. I am using an ugly, but noticeable, **3px** wide border around each section.

_navbar.cshtml | site.css | ProductDAO.cs

```
1  /* Please see documentation at https://docs.
2  for details on configuring this project to b
3
4  header {
5      border:3px dashed red;
6  }
7
8  main {
9      border:3px dashed green;
10 }
11
12 a.navbar-brand {
13     white-space: normal;
14     text-align: center;
15     word-break: break-all;
16 }
17
18 /* Provide sufficient contrast against white
```

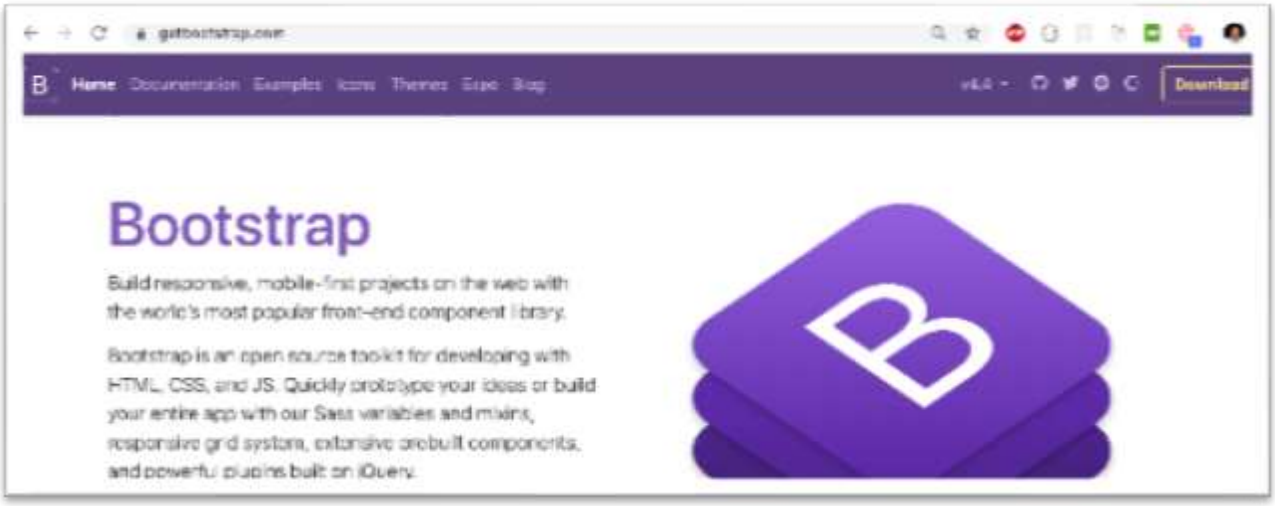
3. Run the application and verify that the sections are displayed. You may have to press **shift while reloading the page** on the browser to force the new CSS changes to appear.

4. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Introduction to Bootstrap CSS

Bootstrap is a popular CSS library that provides you with many pre-defined styles. You should use Bootstrap or some similar library when you are trying to save time in creating a functional website. Bootstrap 4 is the de facto CSS framework for creating a quick design.

The official website for Bootstrap is shown below. The documentation and examples on the official website are very clearly written.



A good tutorial for using Bootstrap is shown here at w3schools.

You will notice that links to Bootstrap CSS and Bootstrap JavaScript are included in the **_Layout.cshtml** file. That means that every page in the application is styled using Bootstrap.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>@ViewData["Title"] - ASPCoreFirstApp</title>
7      <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
8      <link rel="stylesheet" href="~/css/site.css" />
9  </head>
10 <body>
11     <header>
12         @Html.Partial("_navbar.cshtml");
13     </header>
14     <div class="container">
15         <main role="main" class="pb-3">
16             @RenderBody()
17         </main>
18     </div>
19
20     <footer class="border-top footer text-muted">
21         <div class="container">
22             &copy; 2020 - ASPCoreFirstApp - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
23         </div>
24     </footer>
25     <script src="~/lib/jquery/dist/jquery.min.js"></script>
26     <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
27     <script src="~/js/site.js" asp-append-version="true"></script>
28     @RenderSection("Scripts", required: false)
29 </body>
30 </html>
31
```

Bootstrap class names are used in nearly every Razor page in the application. For example, **SearchForm.cshtml**, created in a previous tutorial, has Bootstrap CSS class names in nearly every line of the form. <u>Knowing Bootstrap class names is a key skill to being able to style an ASP.NET application.</u>

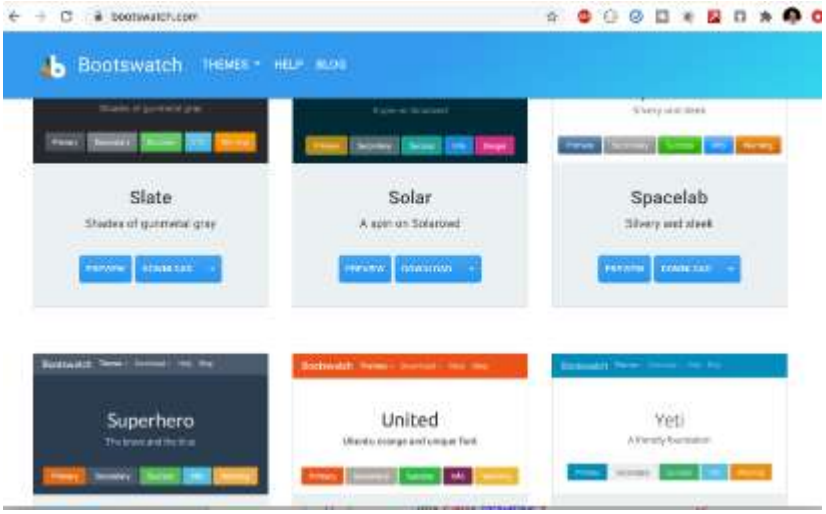
SearchForm.cshtml | navbar.cshtml | site.css | ProductModel.cs | Index.cs

```
1  <h4>Search From</h4>
2  <hr />
3  <div class="row">
4      <div class="col-md-4">
5          <form asp-action="SearchResults">
6
7              <div class="form-group">
8                  <label class="control-label">Search for a Product</label>
9                  <input name="searchTerm" class="form-control" />
10             </div>
11
12             <div class="form-group">
13                 <input type="submit" value="Search" class="btn btn-primary" />
14             </div>
15
16         </form>
17     </div>
18 </div>
19
20 <div>
21     <a asp-action="Index">Back to List</a>
22 </div>
```

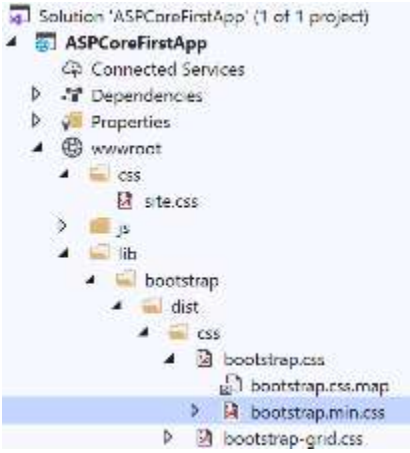
Bootstrap class names.

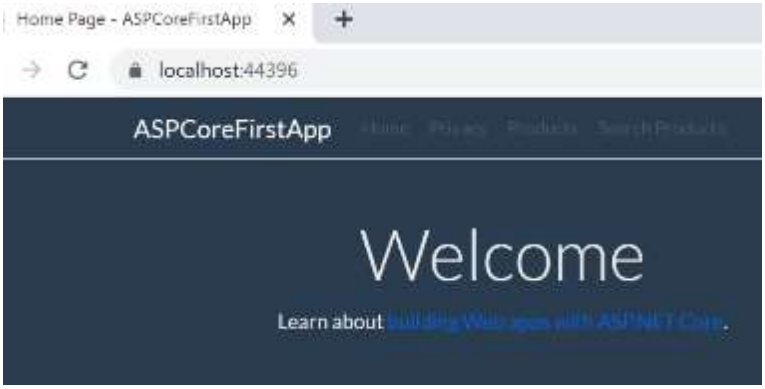
Bootswatch Themes

It is fairly easy to change the theme in an ASP.NET project. Sites like **Bootswatch** provide customized versions of the boostrap.min.css files.

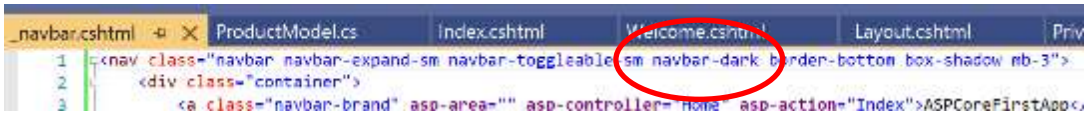


1. Download a theme from **Bootswatch**. You should receive a file called **bootstrap.min.css**
2. Place the file in the **wwwroot > lib > bootstrap > dist > css** folder. You will have to **replace** the existing file.
3. Run the application and refresh the page. You may have to hold the **shift key** while **refreshing** to see the new CSS changes.





4. The CSS class name in the navbar is changed from **navbar-light** to **navbar-dark** to blend better with the chosen theme, *Superhero*.



5. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Alternatives to Bootstrap

Bootstrap may be the most popular of the CSS frameworks, but there are dozens of CSS frameworks available online. However, the pages autogenerated by Visual Studio for an ASP.NET project contain references to Bootstrapp CSS class styles. You will have to change all these class names manually in order to use an alternative CSS framework. Here are three alternatives to Bootstrap to investigate.

1. **Foundation by Zurb**

Second only to Bootstrap in popularity. Foundation has most of the features of Bootstrap.

2. **MaterializeCSS**

Created and designed by Google, Material Design is a design language that combines the classic principles of successful design along with innovation and technology. Google's goal is to develop a system of design that allows for a unified user experience across all their products on any platform.

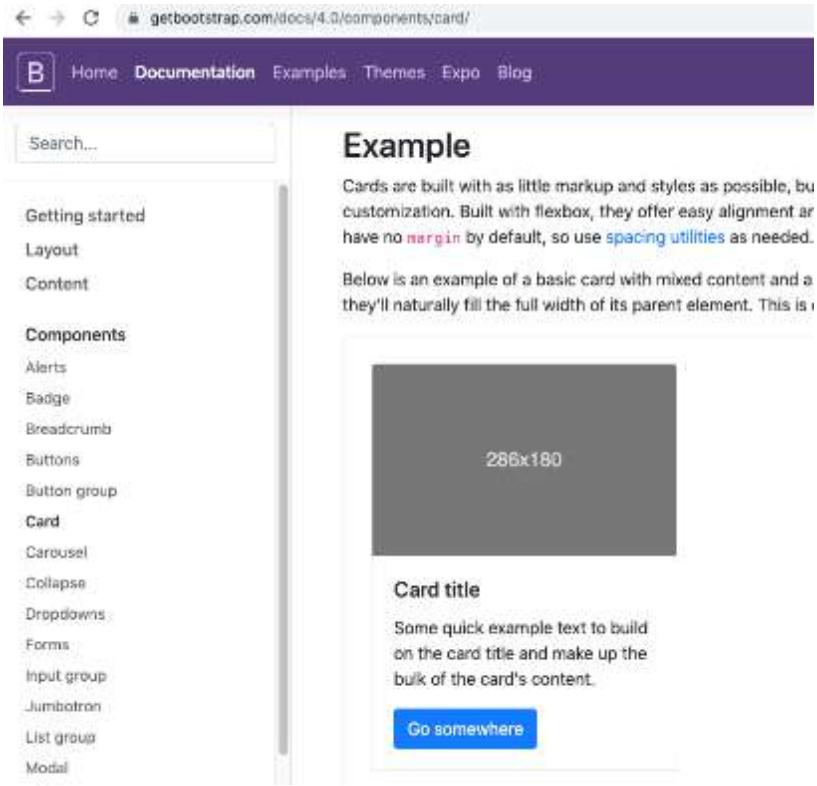
3. **PureCSS**

Known for being extremely small file sizes to reduce webpage load time. Minimal options, but perhaps more than enough to get the basics done.

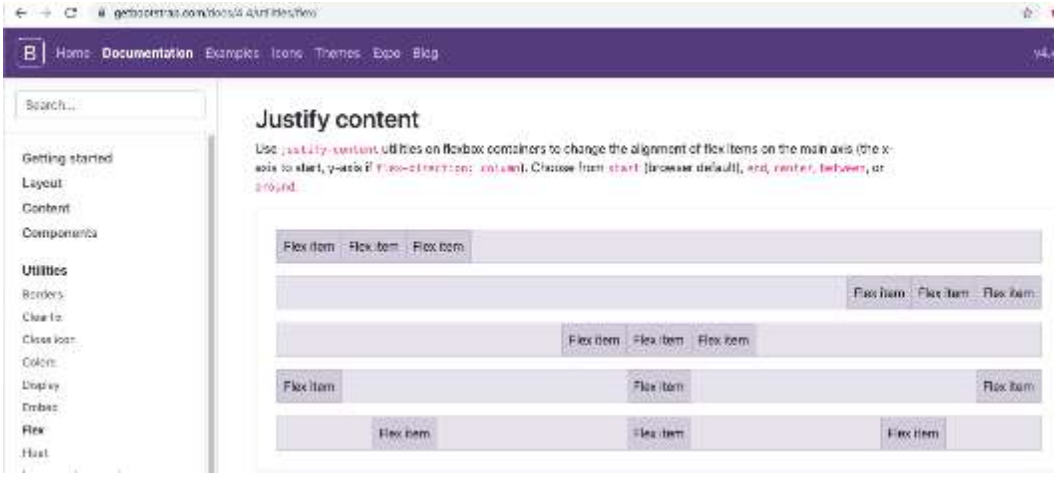
Change a Table to a Flexbox container of cards

In this section, we will replace the table display with a flex box container full of Card objects that looks like the image below. Everything you need to complete the challenge is found in these pages:

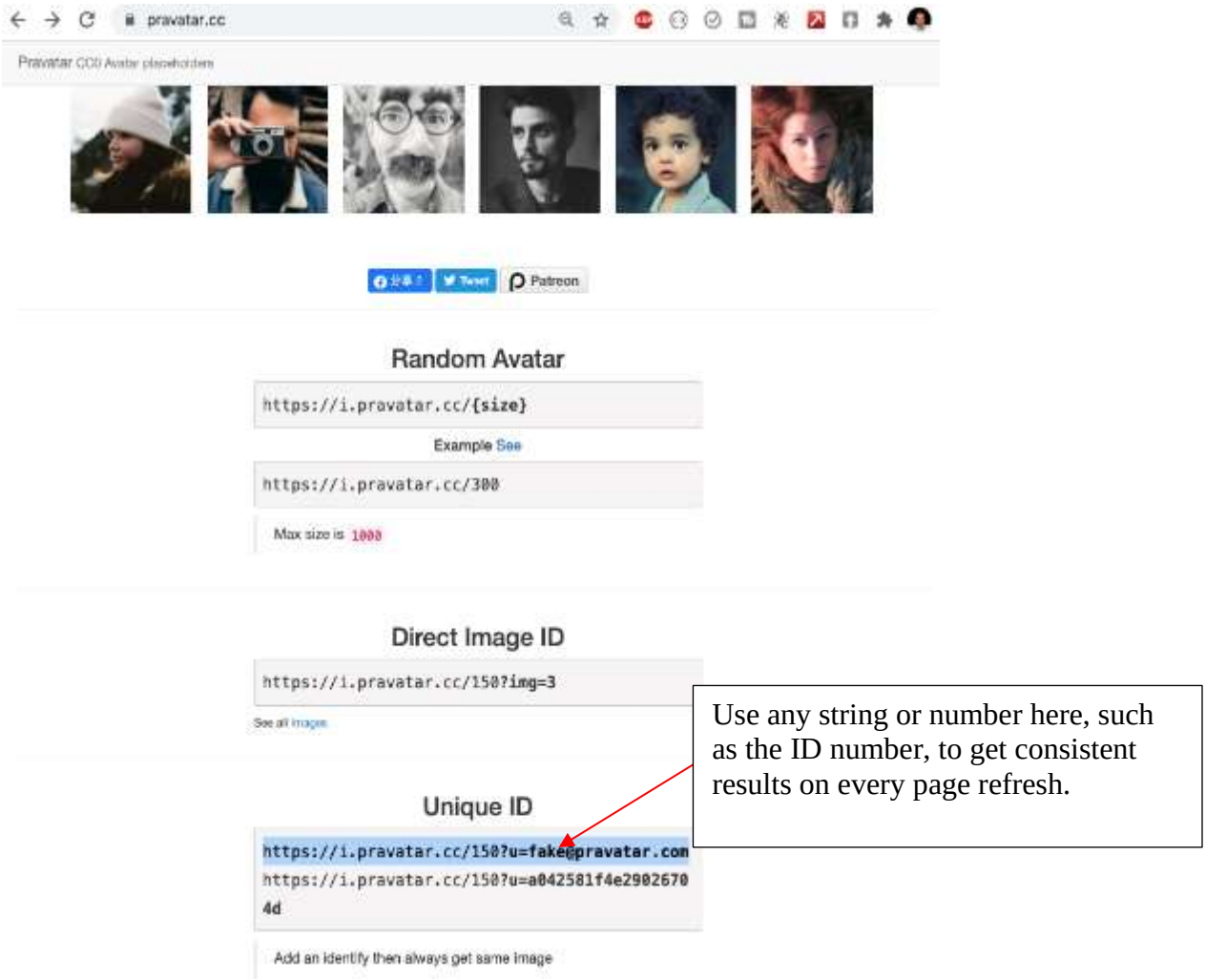
Boostrap cards are described on their site as shown here.



Also shown is how Boostrap uses Flex box CSS styling to arrange items within a container.



Pravatar is a free service that will help you generate random avatar photos for mock data. Our tutorial app uses products, not avatars, but the effect will still be noticeable.



1. Modify the **index.cshtml** view page in the products folder to use:
   - a flexbox container

- a set of Bootstrap cards instead of a table
- Pravatar images

Flexbox container div

A set of bootstrap cards instead of a table

Pravatar images

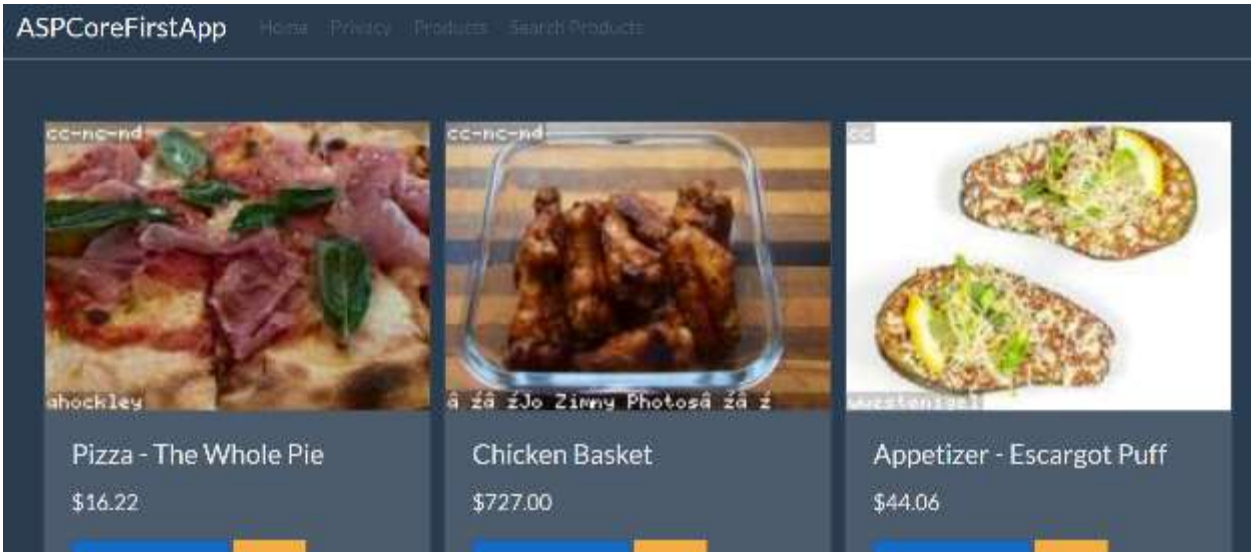
```
@model IEnumerable<ASPCoreFirstApp.Models.ProductModel>

<div class="container d-flex flex-wrap">
    @foreach (var item in Model)
    {
    <div class="card" style="width: 18rem; margin:5px;">
        <img src="https://i.pravatar.cc/150?u=@item.Id" class="card-img-top">
        <div class="card-body">
            <h5 class="card-title">@item.Name</h5>

            <p class="card-text"> @Html.DisplayFor(modelItem => item.Price)</p>
            <a href="#" class="btn btn-primary">Go somewhere</a>
        </div>
    </div>

    }
</div>
```

2. Modify the layout to use another placeholder service. This time from **Loremflickr**, which uses images from Flickr instead of avatar pics.

Lorem flicker image

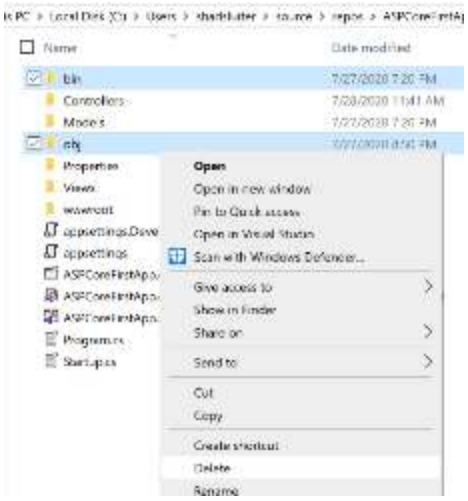
```
@model IEnumerable<ASPCoreFirstApp.Models.ProductModel>

<div class="container d-flex flex-wrap">
    @foreach (var item in Model)
    {
    <div class="card" style="width: 18rem; margin: 5px;">
        @{
            // get just the first word of the item.Name for searching purposes
            var s = item.Name;
            // ternary operator - look it up
            var firstWord = s.IndexOf(" ") > -1 ? s.Substring(0, s.IndexOf(" ")) : s;
        }

        @*look at loremflickr.com for details on how this works*@

        <img src="https://loremflickr.com/160/120/@firstWord?lock=@item.Id" class="card-img-top">
        <div class="card-body">
            <h5 class="card-title">@item.Name</h5>

            <p class="card-text"> @Html.DisplayFor(modelItem => item.Price)</p>
            <a href="#" class="btn btn-primary">Go somewhere</a>
        </div>
    </div>

    }
</div>
```

3. Run the application. You should see a gird of cards with custom pictures.



4. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Deliverables:



1. This activity has multiple parts. Complete all parts before submitting.
2. Submit a Word document with screenshots of the application being run at each stage of development. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
4. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.

Part 3 CRUD SQL operations

Goals:

In this activity, we will return to the Products app and complete the remaining features in the app – **Create, Get One Item, Update, and Delete**.

You will need the code from a previous activity, the **Products Application**.

Show One Item

The first feature that we will complete is the **show one product** on its own **details** page.

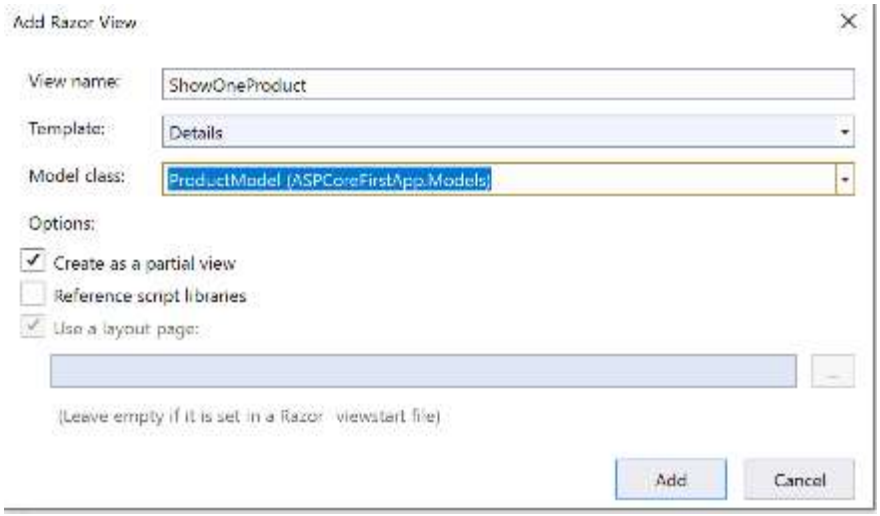
1. In the **ProductDAO** file, implement the GetProductById method as shown here.

```
115     public ProductModel GetProductById(int id)
116     {
117         ProductModel foundProduct = null;
118
119         // uses prepared statements for security. @username @password are defined below
120         string sqlStatement = "SELECT * FROM dbo.Products WHERE Id = @id";
121
122         using (SqlConnection connection = new SqlConnection(connectionString))
123         {
124             SqlCommand command = new SqlCommand(sqlStatement, connection);
125
126             // define the values of the two placeholders in the sqlStatement string
127             command.Parameters.AddWithValue("@id", id);
128
129
130             try
131             {
132                 connection.Open();
133                 SqlDataReader reader = command.ExecuteReader();
134
135                 while (reader.Read())
136                 {
137                     foundProduct = new ProductModel((int)reader[0], (string)reader[1], (decimal)reader[2], (string)reader
                        [3]);
138
139                 }
140             }
141             catch (Exception ex)
142             {
143                 Console.WriteLine(ex.Message);
144             };
145         }
146         return foundProduct;
147     }
```

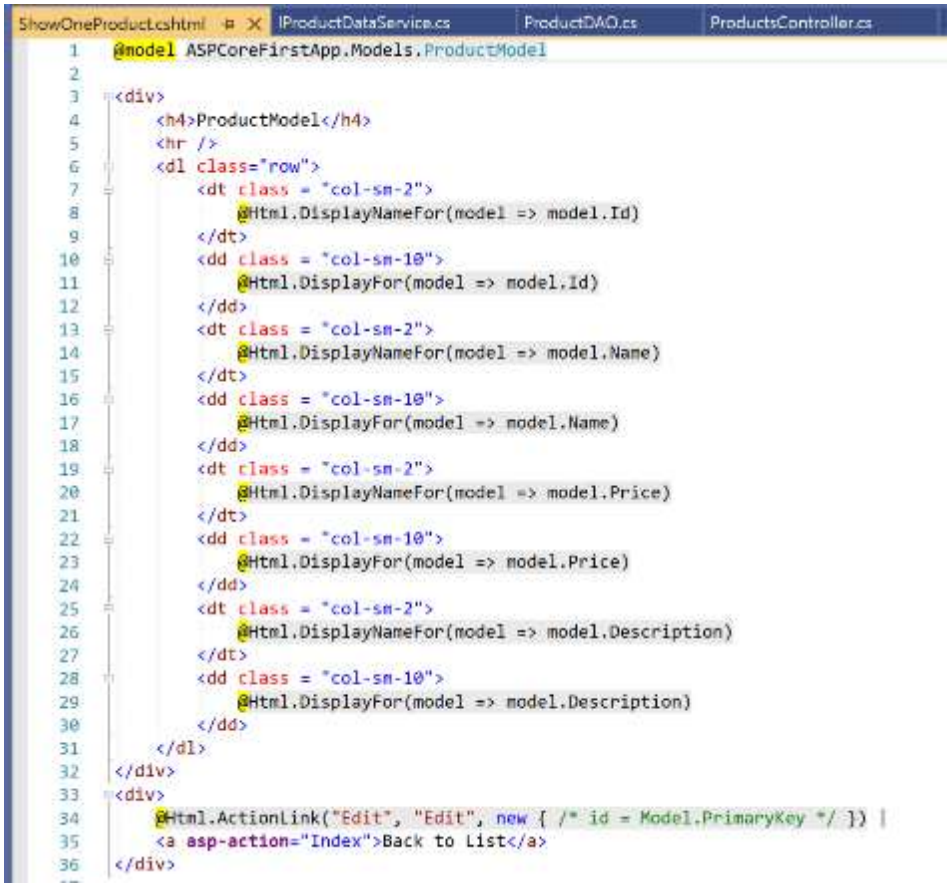
2. In the **ProductsController**, add the method **ShowOneProduct** as shown.

```
36
37      public IActionResult ShowOneProduct(int Id)
38      {
39          return View(repository.GetProductById(Id));
40      }
41
```

3. **Right-click** inside the ShowOneProduct method and choose **Add > View**. Create a new Razor View with the Details template and the ProductModel.



You should see the following code in the new file that displays the default code for details of a single product record.




```
@model ASPCoreFirstApp.Models.ProductModel

<div>
    <h4>ProductModel</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Id)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Id)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Description)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Description)
        </dd>
    </dl>
</div>
<div>
    @Html.ActionLink("Edit", "Edit", new { /* id = Model.PrimaryKey */ }) |
    <a asp-action="Index">Back to List</a>
</div>
```

4. In the **Index.cshtml** file, modify the <a> tag inside the product card. This will direct the browser to display the new view we just created.

```
@model IEnumerable<ASPCoreFirstApp.Models.ProductModel>


<div class="container d-flex flex-wrap">
    @foreach (var item in Model)
    {
    <div id="prod-card-@item.Id" class="card" style="width: 18rem; margin: 5px;">
        @{
            // get just the first word of the item.Name for searching purposes
            var s = item.Name;
            // ternary operator - look it up
            var firstWord = s.IndexOf(" ") > -1 ? s.Substring(0, s.IndexOf(" ")) : s;
        }

        @*look at loremflickr.com for details on how this works*@

        <img src="https://loremflickr.com/160/120/@firstWord?lock=@item.Id" class="card-img-top">
        <div class="card-body">
            <h5 class="card-title">@item.Name</h5>

            <p class="card-text"> @Html.DisplayFor(modelItem => item.Price)</p>
            <a href="/products/ShowOneProduct/@item.Id" class="btn btn-primary">Show Details</a>
        </div>
    </div>

    }
</div>
```

Edit an Item

Next, we will create the ability for the user to update any of the items in the product catalog. We need to do the following:

1. Create the appropriate method in the ProductsDAO.
2. Create two event handlers in the Controller, (1) show the edit form (2) process the update.
3. Create a View to show an edit for the product.
4. Create a button on the products page to allow navigation to the edit form.

New Method in Products DAO

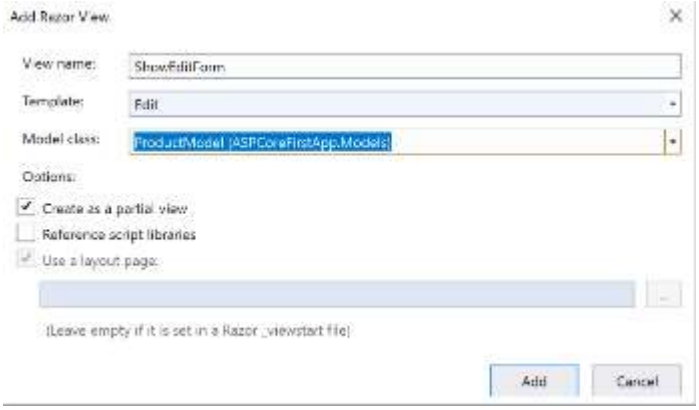
1. In the **ProductsDAO**, implement the **Update** method as shown here.

```
154         public int Update(ProductModel product)
155         {
156             //  Returns -1 if the update fails.
157             int newIdNumber = -1;
158             using (SqlConnection connection = new SqlConnection(connectionString))
159             {
160                 string query = "UPDATE dbo.Products SET Name = @Name, Price = @Price, Description = @Description WHERE Id =
                     @Id";
161
162                 SqlCommand myCommand = new SqlCommand(query, connection);
163                 myCommand.Parameters.AddWithValue("@Id", product.Id);
164                 myCommand.Parameters.AddWithValue("@Name", product.Name);
165                 myCommand.Parameters.AddWithValue("@Price", product.Price);
166                 myCommand.Parameters.AddWithValue("@Description", product.Description);
167
168
169                 try
170                 {
171                     connection.Open();
172
173                     newIdNumber = Convert.ToInt32(myCommand.ExecuteScalar());
174
175                 }
176                 catch (Exception ex)
177                 {
178                     Console.WriteLine(ex.Message);
179                 };
180                 return newIdNumber;
181             }
182
183         }
184     }
```

2. In the **Products** Controller, add the following two methods: **ShowEditForm** and **ProcessEdit**

```
41
42          public IActionResult ShowEditForm(int Id)
43          {
44              return View(repository.GetProductById(Id));
45          }
46
47          public IActionResult ProcessEdit(ProductModel product)
48          {
49              repository.Update(product);
50              return View("Index", repository.AllProducts());
51          }
52
```

3. Right-click in the **ShowEditForm** method and choose **Add > View**. Choose the **Edit** template with the **ProductModel** for the Model class.



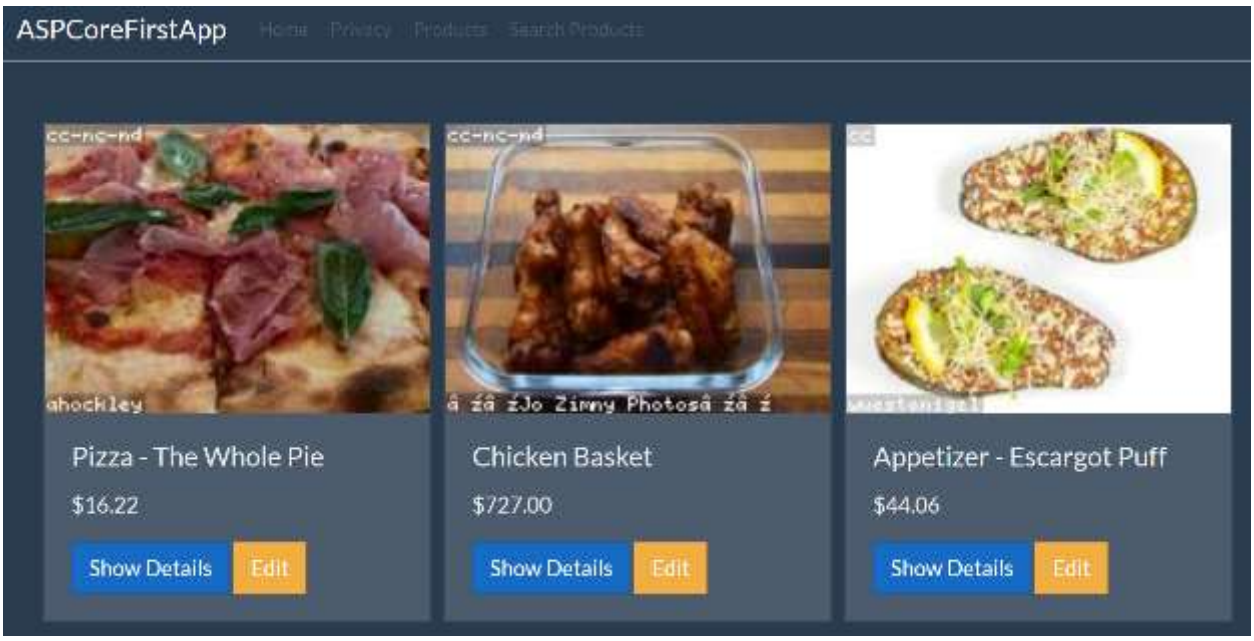
4. Change the form action to **ProcessEdit**

```
@model ASPCoreFirstApp.Models.ProductModel

<h4>ProductModel</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="ProcessEdit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Id" class="control-label"></label>
                <input asp-for="Id" class="form-control" />
                <span asp-validation-for="Id" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Description" class="control-label"></label>
                <input asp-for="Description" class="form-control" />
                <span asp-validation-for="Description" class="text-danger"></spar
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>
```

5. Add another **button** to the **index.cshtml** file to allow for **edits**.

```
@model IEnumerable<ASPCoreFirstApp.Models.ProductModel>


<div class="container d-flex flex-wrap">
    @foreach (var item in Model)
    {
    <div id="prod-card-@item.Id" class="card" style="width: 18rem; margin: 5px;">
        @{
            // get just the first word of the item.Name for searching purposes
            var s = item.Name;
            // ternary operator - look it up
            var firstWord = s.IndexOf(" ") > -1 ? s.Substring(0, s.IndexOf(" ")) : s;
        }

        @*look at loremflickr.com for details on how this works*@

        <img src="https://loremflickr.com/160/120/@firstWord?lock=@item.Id" class="card-img-top">
        <div class="card-body">
            <h5 class="card-title">@item.Name</h5>

            <p class="card-text"> @Html.DisplayFor(modelItem => item.Price)</p>
            <a href="/products/ShowOneProduct/@item.Id" class="btn btn-primary">Show Details</a>
            <a href="/products/ShowEditForm/@item.Id" class="btn btn-warning">Edit</a>
        </div>
    </div>

    }
</div>
```

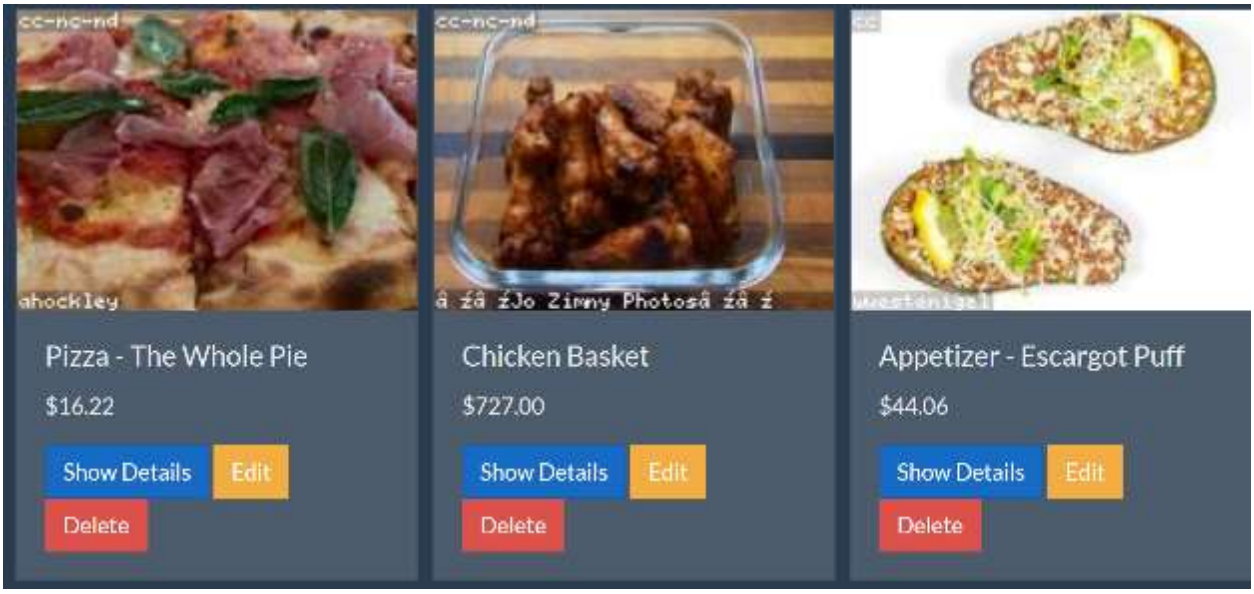
You should be able to run the program, show the **details** of one item and **edit** an item.



Programming Challenge

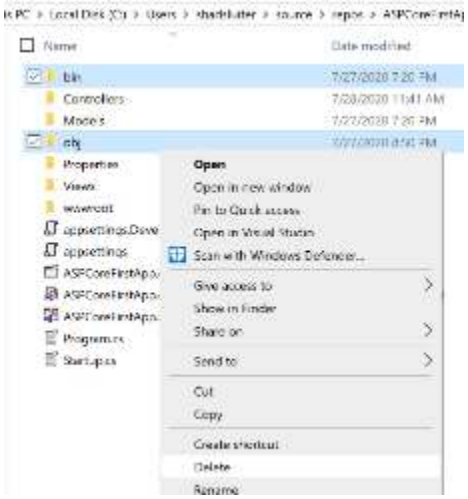
The last database operation to perform is the **delete** command. Add the following changes to the app:

1. Implement the **Delete** method in **ProductsDAO**.
2. Add a **Delete** method to the controller.
3. Add a **Delete** button to the Index view.



For reference materials on SQL statements, see Microsoft's documentation at https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/retrieving-and-modifying-data

Deliverables:



1. This activity has multiple parts. Complete all parts before submitting.
2. Submit a Microsoft Word document with screenshots of the application being run at each stage of development. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
4. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.