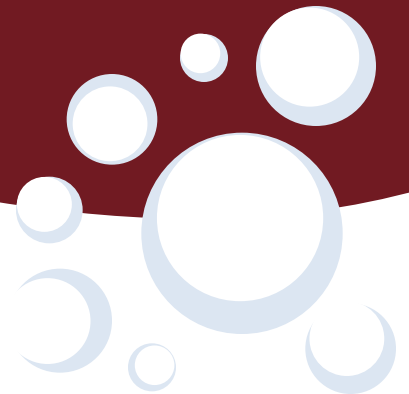


ASP.NET Core 8 Development Lab Manual



Comprehensive ASP.NET Core 8 Development

Visual Studio 2022

About this Lab Manual

This lab manual consists of a series of hands-on lab exercises for learning to build Web UI applications and Web APIs with ASP.NET Core 8 and Visual Studio 2022.

System Requirements

- **Visual Studio 2022** (update 17.9 or later - any edition)
 - You can check the update version number of Visual Studio 2022 by selecting [Help > About Microsoft Visual Studio] from within Visual Studio
 - Visual Studio 2022 Community Edition is available as a free download from <<https://www.visualstudio.com/>>
- **.NET 8 SDK** (8.0.200 or later)
 - Should already be installed as part of the Visual Studio 2022 installation
 - You can check the versions of the SDK that you have installed by executing **dotnet --list-sdks** at a command prompt
 - You can download the SDK separately from <<https://dotnet.microsoft.com/download>>
- **LocalDB or SQL Server** (any version)
 - Installed by default as part of the Visual Studio installation process
 - To check if LocalDB is installed, you can execute **sqllocaldb i** from a command prompt. This should list the LocalDB instances that are available. If the command is not recognized then LocalDB is not installed.
 - If not installed, you can install LocalDB by using the Visual Studio installer via [Tools > Get Tools and Features...] from the Visual Studio menu. LocalDB is listed under "Individual Components" as "SQL Server Express 2016 LocalDB".
 - If using a version of SQL Server other than LocalDB, you must be able to connect to the server with sufficient permissions to create a new database
- An **internet connection** is required to download and install NuGet packages from <<https://api.nuget.org>>

Lab I

Objectives

- Create and run a .NET 8 **console** application using the **CLI**
- Create and run an **ASP.NET Core** application using the **CLI**

Procedures

1. Open a **command prompt** window (or Windows PowerShell).
2. Change the **current directory** to a location where you would like to create some new projects.
3. Use the **dotnet** command to display a list of available **templates** for creating a new project.

```
> dotnet new
```

These are the templates that are included by default. This list can be extended. More information is available at <https://github.com/dotnet/templating>

4. Use the **dotnet** command to create a new **C# console application**.

```
> dotnet new console --name ConsoleApp
```

This command creates a directory for the project and adds two files: a project file (ConsoleApp.csproj) and a source file (Program.cs).

5. Change to the **ConsoleApp** directory and display the contents of **ConsoleApp.csproj**

PowerShell:

```
> cat ConsoleApp.csproj
```

Command Prompt:

```
> type ConsoleApp.csproj
```

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

Within this file, the target framework should be specified as “net8.0”. This is referred to as a Target Framework Moniker (TFM) and specifies the framework that this app is targeting. It is possible to specify more than one TFM to build for multiple targets.

This file is also where references to other packages will be listed. When using Visual Studio, the contents of this file will be managed for you but you can also edit the file manually if the need arises.

6. Examine the contents of the **Program.cs** file.

```
Console.WriteLine("Hello World!");
```

If you have C# development in previous versions of .NET, it may seem like there are several things missing in this file (namespace, class definition, etc.) but this file takes advantage of some new features in C# to reduce the amount of code required. We will talk about these features shortly.

7. Use the **dotnet** command to **restore** the packages that this project depends on.

```
> dotnet restore
```

We didn't specify any additional dependencies in the project file but there are some packages that every .NET application implicitly requires.

The restore command creates a file in the project's obj directory named project.assets.json. This file contains a complete graph of the NuGet dependencies and other information describing the app. This file is what is used by the build system when compiling your app.

8. Use the **dotnet** command to **build** the application.

```
> dotnet build
```

Notice in the output of the build command that the result is a dll file (not an exe file). This is because the dotnet command is used to actually run the application. This is a portable application and this same dll file can be used to run the application on a different platform by using the dotnet command on that platform. In this case, a self-contained exe for Windows is also generated.

9. Use the **dotnet** command to **run** the application and check the output.

```
> dotnet run
```

Note that the run command will automatically invoke “dotnet build” if the application needs to be built.

Now that we have a working .NET 8 console application, we’ll create a new ASP.NET Core 8 application and see what is different and what things are the same.

10. Change the **working directory** back to where you were when you created the console application project.

11. Use the **dotnet** command to create a new project using the **ASP.NET Core Empty (C#)** template.

```
> dotnet new web -n WebApp
```

The Empty template includes the minimum amount of code necessary for a functional ASP.NET Core 8 web application (i.e. Hello World). For future projects, we will use one of the other more full-featured templates.

12. Change the working directory to the **WebApp** directory and examine the list of **files** that were created.

Like the console app, we have a .csproj file, a Program.cs file, and an obj directory. However, we now also have two appsettings files, and a Properties directory.

13. Examine the contents of **WebApp.csproj**.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
</Project>
```

Notice that this file is very similar to the console app. The SDK includes "Web" to implicitly reference some additional packages and the OutputType element is no longer present.

14. Examine the contents of **Program.cs** (part of that file shown below):

```
var builder = WebApplication.CreateBuilder(args);  
var app = builder.Build();  
  
app.MapGet("/", () => "Hello World!");  
  
app.Run();
```

The code here calls a method to create and configure a "HostBuilder" that is built and run. The host runs for the lifetime of our application so that we can respond to incoming HTTP requests.

The MapGet method creates an endpoint for our application that returns the string "Hello World!" when a GET request for the root URL of the application is received.

15. Examine the contents of **launchSettings.json** in the **Properties** directory and look for the **applicationUrl** setting. It may appear more than once.

```
"applicationUrl": "https://localhost:7096;http://localhost:5189",
```

Your port numbers are probably different but this specifies the protocols and ports that the application will use when launched via the CLI.

When using HTTPS, the application will use a self-signed development certificate. Web browsers will not trust that certificate by default. We can use the CLI to configure our machine to trust the development certificate.

16. Make sure you in the same directory as the **csproj** file, **run** the application, and examine the log entries that appear in the console.

```
> dotnet run
```

The log entries in the console should include information about the URLs for the application and the hosting environment (e.g., Development).

17. Open a **web browser** and make a request to the URL shown in the log. You should see the string "Hello World!"

Notice that some logging information also appears in the console when you make a request.

- 18.** Return to the console window and press **ctrl-c** to stop the application.

End of Lab

Lab 2

Objectives

- Create a new ASP.NET Core web project in **Visual Studio**
- **Run** and **test** the application

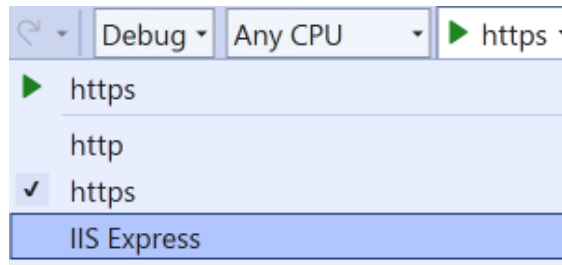
Procedures

1. Open Visual Studio 2022, choose [**File > New > Project...**], select **ASP.NET Core Web App (Model-View-Controller)**, and click **Next**.
 - a. Name the project **EComm.Web**
 - b. Choose an appropriate **location** for the project.
 - c. Change the solution name to be **EComm** and click **Next**
2. In the next dialog, make the following selections:
 - a. Select **.NET 8.0** as the Target Framework
 - b. Select the **None** as the Authentication Type
 - c. Ensure **Configure for HTTPS** is checked
 - d. Ensure **Enable Docker** is **not** checked
 - g. Ensure **Do not use top-level statements** is **not** checked
 - h. Click **Create**
3. **Examine** the files in the solution. The project template includes a single controller for the home page and a privacy page.

If you look in the Framework folder under Dependencies, you can see the two NuGet packages that were mentioned earlier.

4. Expand the **Properties** folder and open the **launchSettings.json** file.

The items within the Profiles element specify different ways that you can launch your application within Visual Studio. In this case, HTTP, HTTPS, or hosted behind IIS Express. These options appear as choices in Visual Studio's debug target list.



The profiles for a project can also be edited by using the Debug tab of the project's properties page (right-click the project node in Solution Explorer and select Properties).

5. **Run** the application with the **http** profile selected. A simple page with a welcome message should appear in the browser.
6. **Close** the browser to **stop** the application.
7. Take a moment to **examine** the code in **HomeController.cs** and the files within the **Views** folder.

End of Lab

Lab 3

Objectives

- Create a **database** with some sample data

Procedures

1. Open a **command prompt** (or PowerShell) and change to the **Labs** directory (where the **EComm.sql** file is located).

The first step in this lab is to execute an SQL script that will create the database that we will use in all of the remaining labs. The instructions assume that you are using the default instance of LocalDB. If this is not the case, modify the server name in the sqlcmd command and remember to modify the server name in the connection string that you will use later in the course.

2. **Execute** the following command:

PowerShell: > sqlcmd -S '(localdb)\MSSQLLocalDB' -i EComm.sql

Command Prompt: > sqlcmd -S (localdb)\MSSQLLocalDB -i EComm.sql

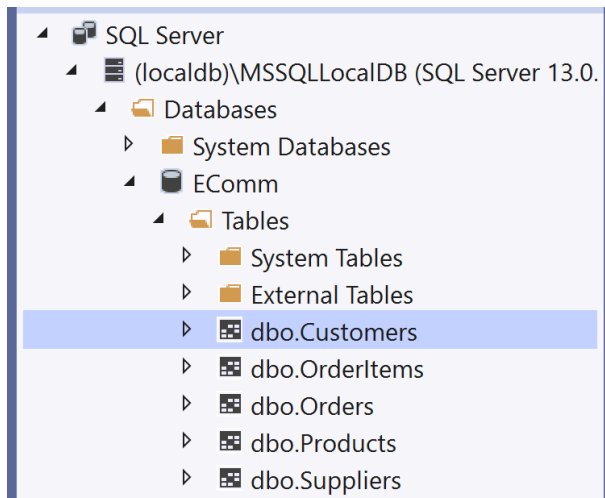
3. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

The database should have been created and populated with some data. To confirm that, we'll view some of the data using SQL Server Object Explorer.

4. From the Visual Studio menu, select [**View > SQL Server Object Explorer**] and expand the **SQL Server** node.
5. If you already have an entry for **(localdb)\MSSQLLocalDB**, skip to **step 6**.
 - a. To add LocalDB, right-click on the SQL Server node and select **Add SQL Server...**
 - b. Expand the **Local** node, select **MSSQLLocalDB**, and click **Connect**.

6. Right-click on **(localdb)\MSSQLLocalDB** in SQL Server Object Explorer, click **Refresh**, and then expand the **EComm** database.



7. Right-click on the **Products** table of the new **EComm** database and select **View Data**. You should see a collection of product records.
8. Take a moment to **explore** the rest of the database tables that are present. There are also two **stored procedures** in the **Programmability** folder.

End of Lab

Lab 4

Objectives

- Add a class library for the **entity types** and **data access abstraction**
- Add a class library for the **data access implementation**

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

2. From the Visual Studio menu, select [**File > Add > New Project...**], choose the **Class Library** template, name the project **EComm.Core**, and select **.NET 8.0**

3. **Delete** the **Class1.cs** file.

The next step is to add a collection of entity objects that match up with the database data that we have. We could manually type in the code for these or use the EF tools to generate the code. For this lab, we will add some files that have already been created.

4. Right-click on the **EComm.Core** project, select [**Add > Existing Item...**], and add the six **.cs** files from the **Lab04** folder.
5. Take a moment to **examine** the code for each of the entity types (we will look at IRepository in the next step). Notice that each of these types is defined in the **EComm.Core.Entities** namespace and each type corresponds to a table in the database.

The properties of the entities also use nullable reference types to match how the columns are defined in the databases (null vs. not null).

6. **Examine** the methods listed in the **IRepository** interface.

Notice that all of the IRepository methods are defined to be asynchronous (each returns a Task object). This will allow us to use the C# await keyword with each these methods.

Some methods have an optional boolean parameter that allows the caller to specify if related entities should be retrieved.

7. Right-click on the **Dependencies** node in the **EComm.Web** project, choose [**Add Project Reference...**], check the box next to **EComm.Core**, and click **OK**.
8. Add another .NET 8.0 class library project named **EComm.Infrastructure** and delete `Class1.cs`

This class library project will contain an `IRepository` implementation but can also be used for any other technology-specific service implementations. For example, a component that can send e-mail messages.

9. Add a project reference to `EComm.Infrastructure` so that it references the **EComm.Core** project.
10. Add a **new class** to the `EComm.Infrastructure` project named **RepositoryEF** that implements the **IRepository** interface.

```
namespace EComm.Infrastructure;

internal class RepositoryEF : IRepository
{
}
```

11. Position your cursor in "`IRepository`", use the pop-up in Visual Studio and select **Implement interface**.

This should have added an implementation of each method defined by `IRepository` with a placeholder of "`throw new NotImplementedException();`". We will add a functional implementation for each method as we need them.

12. **Build** the solution and address any compiler errors/warnings.

End of Lab

Lab 5

Objectives

- Create an **EF-based** implementation of **IRepository**

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

Before we can add any EF-specific code to the infrastructure project, we need to add a package reference for EF Core.

2. Right-click on the EComm.Infrastructure project, select [**Manage NuGet Packages...**] and ensure the **Browse** tab is selected.
3. Search for and install the latest stable version of **Microsoft.EntityFrameworkCore.SqlServer**. Make sure you select the package for EntityFrameworkCore and not EntityFramework (the one for .NET Framework).

Once you click through the installation dialogs, you should see the package listed in the project under Dependencies/Packages.

4. Modify the **RepositoryEF** class so that it inherits from **DbContext**. You will need to add a using directive for Microsoft.EntityFrameworkCore.

```
internal class RepositoryEF : DbContext, IRepository
```

5. Add a **constructor** to RepositoryEF that accepts a **connection string** and stores it into a private field.

```
internal class RepositoryEF : DbContext, IRepository
{
    private readonly string _connStr;

    public RepositoryEF(string connStr)
    {
        _connStr = connStr;
    }
}
```

The `OnConfiguring` method of the `DbContext` can be used to configure the context with the information from the connection string.

6. Override the **OnConfiguring** method of the `DbContext` and use the **UseSqlServer** extension method.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer(_connStr);
}
```

7. Add a **DbSet** property for each of the entity types in our application.

```
public DbSet<Customer> Customers => Set<Customer>();
public DbSet<Order> Orders => Set<Order>();
public DbSet<OrderItem> OrderItems => Set<OrderItem>();
public DbSet<Product> Products => Set<Product>();
public DbSet<Supplier> Suppliers => Set<Supplier>();
```

These `DbSet` properties are necessary so that Entity Framework can figure out what our entities are and provide a way to access them. The properties do have to be public but they are not part of the `IRepository` interface. Our interface methods will be used to encapsulate and control access to these EF-specific details.

8. Implement the **GetAllProducts** method by returning the data provided by Entity Framework. We want to do this in an **asynchronous** way.

```
public async Task<IEnumerable<Product>> GetAllProducts(...)
{
    return includeSuppliers switch {
        true => await Products.Include(p => p.Supplier).ToListAsync(),
        false => await Products.ToListAsync()
    };
}
```

The code above uses a switch expression (introduced in C# 8) but an if statement could have been used instead.

9. Implement the **GetProduct** method in a similar way.

```
public async Task<Product?> GetProduct(int id, bool ...)
{
    return includeSupplier switch {
        true => await Products.Include(p => p.Supplier)
            .SingleOrDefaultAsync(p => p.Id == id),
        false => await Products.SingleOrDefaultAsync(p => p.Id == id)
    };
}
```


We will implement the other methods of RepositoryEF later on as they are needed.

10. Build the solution and address any **errors** or **warnings** that appear.

End of Lab

Note that RepositoryEF does not include any error handling or logging (e.g., what if the database is unreachable?). This will be addressed later in the course.

Lab 6

Objectives

- Register a **service**
 - Modify a controller to accept a **dependency**
 - Return a response that includes **database data**
-

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

In a previous lab, we defined an interface for our data access component and created an implementation that uses Entity Framework Core. In this lab, we will register that component as an application service and use it to return data from a controller.

2. Right-click on the **Dependencies** node in the **EComm.Web** project and add a project reference for **EComm.Infrastructure**

To create an instance of our data access component, we will need to supply a database connection string. A good place for this to be stored is in the `appsettings.json` file. This will allow us to easily specify a different connection string in `appsettings.Development.json`.

3. Open the **appsettings.json** for editing and add an entry for the connection string.
 - a. Make sure to add a **comma** at the end of the line for "AllowedHosts".
 - b. The connection string below is shown on two lines but it must all be on **one line** in the JSON file.

```
"AllowedHosts": "*,  
"ConnectionStrings": {  
  "ECommConnection": "Data Source=(localdb)\\MSSQLLocalDB;  
                      Initial Catalog=EComm;Integrated Security=True"  
}
```

At this point, we would like to configure our RepositoryEF class as a service in our application. However, the accessibility of the RepositoryEF class is internal. This is intentional to support type encapsulation but our application needs a way to obtain an instance.

4. Add code in **RepositoryEF.cs** that implements a simple **factory**.

```
public static class RepositoryFactory
{
    public static IRepository Create(string connStr) =>
        new RepositoryEF(connStr);
}

internal class RepositoryEF : DbContext, IRepository
...

```

5. Open **Program.cs** in EComm.Web and find where the services are added.
6. Add code to register **IRepository** as a service with our factory method.

```
var connStr = builder.Configuration.GetConnectionString("ECommConnection");
if (connStr == null) {
    throw new Exception("Missing connection string");
}
else {
    builder.Services.AddScoped<IRepository>(sp =>
        RepositoryFactory.Create(connStr));
}

```

With this code in place, whenever the DI system is asked for an instance of IRepository, the DI system will invoke the factory method if a new instance is required.

7. Open **HomeController.cs** and add a **private field** of type **IRepository**.

```
public class HomeController : Controller
{
    private readonly IRepository _repository;
    private readonly ILogger<HomeController> _logger;
}

```

8. Modify the **constructor** of HomeController so that it accepts an object that implements IRepository and stores it in the private field.

```
public HomeController(IRepository repository,
    ILogger<HomeController> logger)
{
    _repository = repository;
    _logger = logger;
}

```

9. Modify the **Index** action of **HomeController** so that it is asynchronous and returns some model information.

```
public async Task<IActionResult> Index()
{
    var products = await _repository.GetAllProducts();
    return Content($"Number of products: {products.Count()}");
}
```

10. Run the application and check the results. If a `SqlException` is thrown, check the accuracy of the connection string in `appsettings.json`.
11. Stop the application and add code to the **OnConfiguring** method of **RepositoryEF** to enable some **simple logging**.

```
protected override void OnConfiguring(DbContextOptionsBuilder ...)
{
    optionsBuilder.UseSqlServer(_connStr);
    optionsBuilder.LogTo(Console.WriteLine);
}
```

12. Run the application and check the **console** after executing the action. You should be able to find the SQL that was sent to the database.

```
Executed DbCommand (26ms)
[Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [p].[Id], [p].[IsDiscontinued], [p].[Package],
[p].[ProductName], [p].[SupplierId], [p].[UnitPrice]
FROM [Products] AS [p]
```

This SQL makes sense because the `GetAllProducts` method returns all of the products as a list. However, it's not efficient as it could be since we are only interested in the count. We could add another method for returning only the count but that is not necessary here (this was just a test - we won't be retrieving only the count in the final application).

13. As a final experiment, modify the **Index** action to return all of the products as **JSON**.

```
var products = await _repository.GetAllProducts();
return Json(products);
```

14. Run the application and check the results.

Depending on the browser you are using, the JSON may render directly into the browser window or the browser may prompt you to download the data as a file.

End of Lab

Lab 7

Objectives

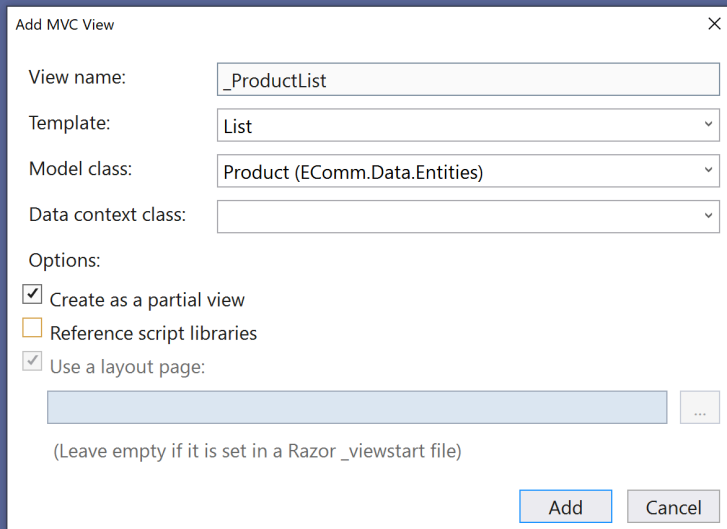
- Use a **partial view** to display the **list of products**

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

2. In **EComm.Web**, right-click on the **Views/Shared** folder, choose [**Add > View...**], select **Razor View**, click **Add**, make the selections as shown below, and click **Add**. (the namespace for the Product type may be different)



We want this to be a partial view but we don't need to reference the script libraries. That is used to add client-side data validation (JavaScript). Since we selected List as the template and Product as the model class, this view will be strongly-typed to a collection of products and so we will need to pass it a collection of products.

3. Open **Index.cshtml** for editing and add a **partial tag helper** at the bottom of the page (below `</div>`)

```
<partial name="_ProductList" />
```

Since the partial view needs a list of products, the view that uses the partial view needs to receive a list of products. The model of the parent view will automatically be available to the partial view without the need to explicitly pass it.

4. Add a line to the very top of **Index.cshtml** that makes the view **strongly-typed** to a **collection of products**.

```
@model IEnumerable<EComm.Core.Entities.Product>
@{
    ViewData["Title"] = "Home Page";
}
```

5. Open **HomeController.cs** and modify the **Index** action so that it passes the **collection of products** to the view.

```
public async Task<IActionResult> Index()
{
    var products = await _repository.GetAllProducts();
    return View(products);
}
```

6. **Run** the application and check the results. The product list on the home page should look like the image below.

[Create New](#)

Id	ProductName	UnitPrice	Package	IsDiscontinued	SupplierId	
1	Chai	18.00	10 boxes x 20 bags	<input type="checkbox"/>	1	Edit Details Delete
2	Chang	19.00	24 - 12 oz bottles	<input type="checkbox"/>	1	Edit Details Delete
3	Aniseed Syrup	10.00	12 - 550 ml bottles	<input type="checkbox"/>	1	Edit Details Delete
4	Chef Anton's Cajun Seasoning	22.00	48 - 6 oz jars	<input type="checkbox"/>	2	Edit Details Delete
5	Chef Anton's Gumbo Mix	21.35	36 boxes	<input checked="" type="checkbox"/>	2	Edit Details Delete
6	Grandma's Boysenberry Spread	25.00	12 - 8 oz jars	<input type="checkbox"/>	3	Edit Details Delete
7	Uncle Bob's Organic Dried Pears	30.00	12 - 1 lb pkgs.	<input type="checkbox"/>	3	Edit Details Delete

This is the list of products from the database but there are few things that should be addressed. We will work on some of those things now.

- a. The **Id** column **should not** be displayed in this list.
- b. The **Supplier** column should show the **name of the supplier** and not the Id.
- c. The **links** on the right side do not work yet.
- d. Some other display issues like the text of the **column headers** and maybe formatting Unit Price as currency.

7. Open **_ProductList.cshtml** and remove the **Id** column (the first `<th>...</th>` in the `<thead>` element and the first `<td>...</td>` in the `<tbody>` element).
8. Change the **SupplierId** header to display the name of the **Supplier** property.

```
<th>
    @Html.DisplayNameFor(model => model.Supplier)
</th>
```

9. Change the value displayed in the **Supplier** column from **SupplierId** to the **CompanyName** of the supplier.

```
<td>
    @Html.DisplayFor(modelItem => item.Supplier!.CompanyName)
</td>
```

The null-forgiving operator (!) is used here since the Supplier property is nullable but we know that it will have a value at this point. If this was not the case, we could add code to specify what to display if null (e.g., "Unknown").

10. Run the application and notice that the Supplier column is **blank**.

The reason the Supplier column is blank is because EF does not fetch related objects like supplier unless we specifically request that behavior (eager loading). Our ECommDataContext does offer that capability via the includeSuppliers optional parameter.

11. Modify the call to **GetAllProducts** in HomeController to pass true for the **includeSuppliers** parameter.

```
var products = await _repository.GetAllProducts(includeSuppliers: true);
return View(products);
```

12. Run the application and check that the **company name** of each supplier is being displayed. Also look at the **log** and examine the **SQL** that EF is using to get the product data. Is it executing a separate query to get the supplier for each product or is it using a single query?

End of Lab

Lab 8

Objectives

- Add the ability to display the **details** for a product

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

*If you do not want to use your solution from the previous lab, you can open the solution in the **Begin** folder for this lab.*

2. Use what you have learned to enable the Details link for the list of products. A list of requirements is provided below. Step-by-step instructions are provided on the next page but you should try your best to accomplish the task without using those instructions.
 - a. There should be a new controller named **ProductController** with an action named **Details** that has a route of **product/{id}**
 - b. Clicking the **Details link** in a product's row should invoke the Details action and return the details for an individual product.
 - c. You can use the **Details template** when creating the view.
 - d. Like the product list, the **company name** of the supplier should be shown on the details page instead of the supplier id.
 - e. Make sure the **"Back to List"** link on the **Details** page works properly.

3. Right-click on the Controllers folder in EComm.Web, choose [**Add > Controller...**], select the **MVC Controller - Empty** template, click **Add**, and name the controller **ProductController**
4. Delete the **Index** action and add the same **private fields** and **constructor** that HomeController has.

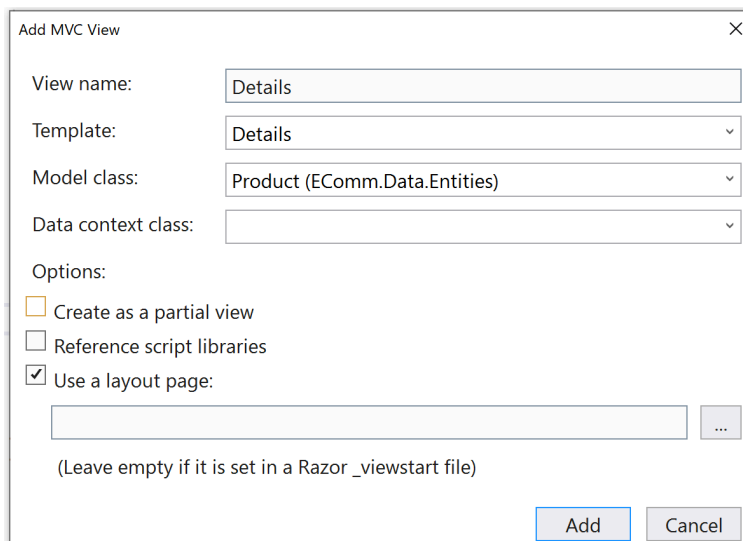
```
public class ProductController : Controller
{
    private readonly IRepository _repository;
    private readonly ILogger<ProductController> _logger;

    public ProductController(IRepository repository,
                           ILogger<ProductController> logger)
    {
        _repository = repository;
        _logger = logger;
    }
}
```

5. Add a **Details** action to the **ProductController** with an appropriate **route**, takes an **id** parameter, fetches the correct **product**, and passes it to a **view**.

```
[HttpGet("product/{id}")]
public async Task<IActionResult> Details(int id)
{
    var product = await _repository.GetProduct(id, includeSupplier: true);
    return View(product);
}
```

6. Right-click somewhere in the code for the **Details** action, select [**Add View...**], choose **Razor View**, click **Add**, and make the selections shown below (namespace for Product may be different):



View name: Details

Template: Details

Model class: Product (EComm.Data.Entities)

Data context class:

Options:

☐ Create as a partial view

☐ Reference script libraries

☒ Use a layout page:

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

Add Cancel

By using the right-click menu in the action, Visual Studio should automatically put the view file into the Views/Product directory.

7. In **Details.cshtml**, delete the first **<dt>** element and the first **<dd>** element (that are used to display the name and value of the **Id** property).
8. Find the last **<dt>** and **<dd>** elements (used to display the **SupplierId** property) and **modify** them to display the name of the **Supplier** property and the **CompanyName** of the supplier (similar to what you did earlier for the product list).

```
<dt class = "col-sm-2">
    @Html.DisplayNameFor(model => model.Supplier)
</dt>
<dd class = "col-sm-10">
    @Html.DisplayFor(model => model.Supplier!.CompanyName)
</dd>
```

9. Open **_ProductList.cshtml** and find the helper that generates the **Details** link.
10. Modify the helper to invoke the new **Details** action.

```
<td>
    @Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) |
    @Html.ActionLink("Details", "Details", "Product", new { id=item.Id }) |
    @Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ })
</td>
```

Notice that you have to add the controller name to the call to `Html.ActionLink` since the `Details` action is not in `HomeController`.

11. **Run** the application and check the result when a **Details** link is clicked.

Details	
Product	
ProductName	Aniseed Syrup
UnitPrice	10.00
Package	12 - 550 ml bottles
IsDiscontinued	<input type="checkbox"/>
Supplier	Exotic Liquids
Edit Back to List	

Unfortunately, the "Back to List" link on the Details page does not work properly. The link is trying to go to the Index action of ProductController when, in our application, it should go to the Index action of HomeController.

- I2.** In **Details.cshtml**, add the name of the controller to the **tag helper** used to generate the "Back to List" link.

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

- I3. Run** the application and test the link.

End of Lab

Lab 9

Objectives

- Refactor the product list to be a **view component**

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

2. Add a new **top-level folder** to the EComm.Web project named **ViewComponents**
3. Add a **new class** to the ViewComponents folder named **ProductList** that inherits from **ViewComponent**. You will need to add a **using** directive for **Microsoft.AspNetCore.Mvc**

```
public class ProductList : ViewComponent
```

4. Add the same **constructor** and **private fields** used by ProductController so that the view component can receive an **IRepository** via **dependency injection**.

```
public class ProductList : ViewComponent
{
    private readonly IRepository _repository;
    private readonly ILogger<ProductList> _logger;

    public ProductList(IRepository repository,
                      ILogger<ProductList> logger)
    {
        _repository = repository;
        _logger = logger;
    }
}
```

5. Add an **InvokeAsync** method to the view component that retrieves the **list of products** and sends them to a **view**.

```
public async Task<IViewComponentResult> InvokeAsync()
{
    var products = await _repository.GetAllProducts(includeSuppliers: true);
    return View(products);
}
```

We are now passing data to a view that does not yet exist.

6. Create a **folder** under the **Views/Shared** folder named **Components**.
7. Create a **folder** under the new **Components** folder named **ProductList**.
8. Move the **_ProductList.cshtml** file into the new **ProductList** folder (you can just drag it to the new location).
9. Rename **_ProductList.cshtml** to **Default.cshtml**

The product list view component is now ready to be used in place of the partial view.

10. Open **Index.cshtml** and **replace** the **partial view** with a call to the **ProductList** view component.

```
@await Component.InvokeAsync("ProductList")
```

*Since the view component can fetch the data that it needs on its own, we no longer need **Index.cshtml** to be strongly-typed.*

11. **Delete** the first line of **Index.cshtml** that starts with **"@model"**.
12. Open **HomeController.cs** and update the **Index** action to reflect the fact that we no longer need to fetch and pass a **list of products**.

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

*Notice that this action no longer needs to be asynchronous since we are not awaiting on any operation here. At this point, there are no actions in **HomeController** that use **IRepository** so we could remove that code as well. However, we may add code later that does use **IRepository** and so we'll leave it alone for now.*

13. **Run** the application and confirm that everything works the same as before.

*This lab was a refactor of the application. We didn't alter any of the functionality from the user's point of view. However, we did improve the organization of the code by removing a product-related responsibility from **HomeController**. We also made the product list more reusable by making it a view component.*

End of Lab

Lab 10

Objectives

- Create the **product edit form**

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

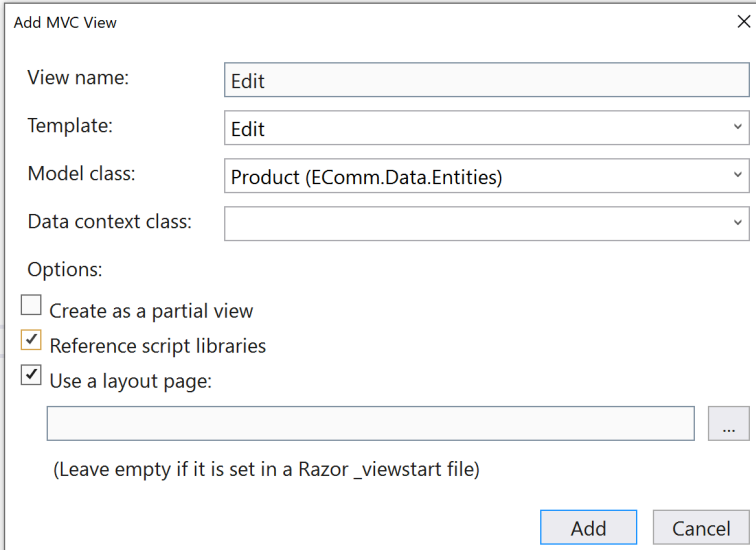
*If you do not want to use your solution from the previous lab, you can open the solution in the **Begin** folder for this lab.*

2. Use what you have learned to add a form for editing a product. A list of requirements is provided below. Step-by-step instructions are provided on the next page but you should try your best to accomplish the task without using those instructions.
 - a. The **ProductController** should have an **Edit** action with a route of "**product/edit/{id}**".
 - b. Use the **Edit template** when adding the **view** and ensure **reference script libraries** is selected (to enable client-side data validation).
 - c. The **SupplierId** property should be represented as a **drop-down list** of **company names**. Create a **view model** (ProductEditViewModel) to provide the collection of SelectListItem objects (and the other data) to the view.
 - d. Ensure that the **Edit links** in the product list take the user to the correct form.
 - e. The form should display the correct data but you do **not** need to handle the submitted data at this point (that will be the next lab).

2. Open **ProductController.cs** and add a new action named **Edit**.

```
[HttpGet("product/edit/{id}")]
public async Task<IActionResult> Edit(int id)
{
    var product = await _repository.GetProduct(id, includeSupplier: true);
    return View(product);
}
```

3. Right-click somewhere in the code for the **Edit** action, select [**Add View...**], choose **Razor View**, click **Add**, and make the following selections (namespace for Product might be different):



View name: Edit

Template: Edit

Model class: Product (EComm.Data.Entities)

Data context class:

Options:

☐ Create as a partial view

☒ Reference script libraries

☒ Use a layout page:

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

Add Cancel

Notice that "Reference script libraries" is checked this time. This will insert JavaScript references into the view so that we can enable client-side data validation in a future lab.

4. Open the **view** for the **product list** (/Views/Shared/Components/ProductList/Default.cshtml) and modify the **Edit link**.

```
@Html.ActionLink("Edit", "Edit", "Product", new { id=item.Id }) |
```

5. **Run** the application and click the Edit link for one of the products and check the edit form. The **Save** button will **not** work yet.

Edit

Product

Id

ProductName

UnitPrice

Package

☐ IsDiscontinued

SupplierId

[Back to List](#)

The form should display the correct data for the selected product. However, there is more work that needs to be done. First, the `Id` property should not be editable (or even displayed). Second, the last form field is `SupplierId`. It should be possible to change the `SupplierId` but the UI should be a drop-down list of company names. Since the form will now need something extra (the list of suppliers to choose from), we should change this view to accept a view model that can provide that information. It will be the job of the controller to fetch the data, construct the view model object, and pass it to the view.

- Open `/Views/Product/Edit.cshtml` for editing and delete the **form-group** used to display the **Id** property (lines 15-19).

Since we just deleted the input for the product's `Id`, we need to make sure the `id` parameter for the `Edit` action is available when the form is submitted. We could use a hidden form field but we will add the `Id` parameter to the action of the form instead.

- Modify the **form tag helper** to include the **Id** of the product being edited.

```
<form asp-action="Edit" asp-route-id="@Model.Id">
```

The next step is to define a view model that can provide the view with all of the data that it needs (including the list of suppliers for the drop-down list).

- Add a new class to the **Models** folder in `EComm.Web` named **ProductEditViewModel** but change it to be a **record** type.

We are adding this class to the `Models` folder in `EComm.Web` because it is a web-specific type that supports a view.

Since `ProductEditViewModel` will act as a data transfer object (DTO), a record type is a good fit for this task.

9. Add the **properties of a Product** to **ProductEditViewModel** plus a property for the collection of suppliers.

```
public record ProductEditViewModel
(
    int Id,
    string ProductName,
    decimal? UnitPrice,
    string Package,
    bool IsDiscontinued,
    int SupplierId,
    Supplier? Supplier,
    IEnumerable<Supplier> Suppliers
);
```

Instead of reimplementing the properties of a Product, we could use inheritance or composition but that would require us to change the tag helpers in the view. The approach we are using also has the advantage of allowing us to remove or rename some properties of a product.

The controller will set the Suppliers properly but the view will need a collection of SelectListItem objects.

10. Add a **read-only property** to **ProductEditViewModel** that provides the **list of suppliers** as a collection of **SelectListItems**. You will need to add a **using** directive.

```
public IEnumerable<SelectListItem> SupplierItems =>
    Suppliers is null ? Array.Empty<SelectListItem>() :
    Suppliers
        .Select(s => new SelectListItem
            { Text = s.CompanyName, Value = s.Id.ToString() })
        .OrderBy(item => item.Text);
```

We are using LINQ's data projection feature to transform the Supplier objects into SelectListItem. There are other ways this can be done.

11. Open **/Views/Product/Edit.cshtml** for editing and change the view to be strongly-typed to a **ProductEditViewModel**

```
@model ProductEditViewModel
```

12. Modify the **<div>** element for **SupplierId** to use **Supplier** as the label, remove the **validation tag helper**, and use the **select tag helper**.

```
<div class="form-group">
    <label asp-for="Supplier" class="control-label"></label>
    <select asp-for="SupplierId" asp-items="@Model.SupplierItems"
        class="form-select"></select>
</div>
```

ProductController will fetch the list of all the suppliers but we have not yet provided an implementation for RepositoryEF's GetAllSuppliers method.

- 13.** Open **RepositoryEF.cs** and add code to retrieve the suppliers via the DbSet.

```
public async Task<IEnumerable<Supplier>> GetAllSuppliers()
{
    return await Suppliers.OrderBy(s => s.CompanyName).ToListAsync();
}
```

- 14.** Open **ProductController.cs** and modify the **Edit** action so that it creates an instance of **ProductEditViewModel** and passes it to the view.

```
[Route("product/edit/{id}")]
public async Task<IActionResult> Edit(int id)
{
    var product = await _repository.GetProduct(id, includeSuppliers: true);
    var suppliers = await _repository.GetAllSuppliers();

    var pvm = new ProductEditViewModel(
        Id: product.Id,
        ProductName: product.ProductName,
        UnitPrice: product.UnitPrice,
        Package: product.Package,
        IsDiscontinued: product.IsDiscontinued,
        SupplierId: product.SupplierId,
        Supplier: product.Supplier,
        Suppliers: suppliers
    );
    return View(pvm);
}
```

We are manually populating the ProductEditViewModel here. This code could be moved into ProductEditViewModel (constructor that takes a Product) or we could use another library to help with this (e.g.AutoMapper).

Visual Studio will show warnings for when we access the product since the product might be null (request is for a product that does not exist). We will eliminate these warnings when we talk about error handling.

- 15. Run** the application and check the appearance of the product edit form. The supplier should be represented by a drop-down box. Of course, the save button still won't work.

End of Lab

Lab 11

Objectives

- Complete the ability to **edit a product**

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

*If you do not want to use your solution from the previous lab, you can open the solution in the **Begin** folder for this lab.*

2. Use what you have learned to handle the submission of the edit form. A list of requirements is provided below. Step-by-step instructions are also provided on the next page but you should try your best to accomplish the task without using those instructions.
 - a. The **ProductController** should have an **Edit** action with an HTTP post route of "**product/edit/{id}**" that accepts a parameter of type **ProductEditViewModel**.
 - b. Check the **ModelState.IsValid** property and return the view again if the property is false (you will need to re-populate the Suppliers property).
 - c. If validation passes (at this point, it always will), **save** the modified product to the database.
 - d. After performing the save, **redirect** the user to the **details view** for the product.
 - e. Make sure the "**Back to List**" link on the Edit page and the "**Edit**" link on the Details page both work properly.

3. Open **ProductController** and add another **Edit** action that can handle the form submission.

```
[HttpPost("product/edit/{id}")]
public async Task<IActionResult> Edit(int id, ProductEditViewModel pvm)
{
    if (!ModelState.IsValid) {
        pvm = pvm with { Suppliers = await _repository.GetAllSuppliers() };
        return View(pvm);
    }
    var product = new Product {
        Id = id,
        ProductName = pvm.ProductName,
        UnitPrice = pvm.UnitPrice,
        Package = pvm.Package ?? string.Empty,
        IsDiscontinued = pvm.IsDiscontinued,
        SupplierId = pvm.SupplierId
    };
    await _repository.SaveProduct(product);
    return RedirectToAction("Details", new { id = id });
}
```

At this point, you should be asking questions about input validation and error handling. We will address those later in the course.

4. Provide an implementation for the **SaveProduct** method of RepositoryEF.

```
public async Task<bool> SaveProduct(Product product)
{
    Products.Attach(product);
    Entry(product).State = EntityState.Modified;
    int rowsAffected = await SaveChangesAsync();
    return (rowsAffected > 0);
}
```

This implementation will update all columns of product but require one trip to the database server. Alternatively, we could fetch the product first and then update it. This would result in a more efficient update operation but would require two trips to the database server.

5. Open **Details.cshtml** and enable the **Edit** link.

```
@Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
```

6. Open **Edit.cshtml** and fix the **"Back to List"** link.

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

7. **Run** the application and test the complete process for editing a product.

End of Lab

Lab 12

Objectives

- Add **data validation** for when editing a product

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

*If you do not want to use your solution from the previous lab, you can open the solution in the **Begin** folder for this lab.*

2. Use what you have learned to implement data validation for the product edit form. A list of requirements is provided below. Step-by-step instructions are also provided on the next page but you should try your best to accomplish the task without using those instructions.
 - a. The product's name should be **required**.
 - b. Unit price should be **required** with a **minimum** value of **1.00** and a **maximum** value of **500.00**
 - c. Customize the **message** provided for each validation rule (can be anything you like).
 - d. Ensure that both **client-side** and **server-side** validation is functioning. One easy way to do this is by commenting-out the inclusion of **_ValidationScriptsPartial.cshtml** in **Edit.cshtml**

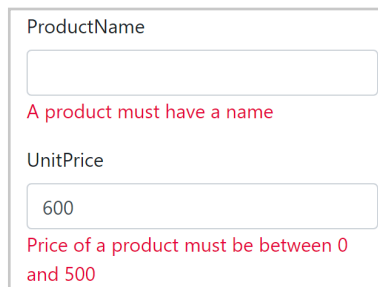
3. In `ProductEditViewModel.cs`, add the **Required** attribute to the **ProductName** property. You will need to add a **using** directive for `System.ComponentModel.DataAnnotations`

```
[Required(ErrorMessage = "A product must have a name")]
string ProductName,
```

4. Add the **Required** and **Range** attribute to the **UnitPrice** property.

```
[Required(ErrorMessage = "A product must have a price")]
[Range(1.0, 500.0,
    ErrorMessage = "Price of a product must be between 1 and 500")]
decimal? UnitPrice,
```

5. **Run** the application and try to edit a product with data that does not pass the validation rules.



ProductName

A product must have a name

UnitPrice

Price of a product must be between 0 and 500

This test is using the client-side validation that is inserted by the tag helpers. We would also like to test the server-side validation logic that is executed during the model binding process. We can do this by removing the JavaScript used to perform the client-side validation.

6. Find the line in **Edit.cshtml** that uses **_ValidationScriptsPartial.cshtml** and comment it out.

```
@* @{await Html.RenderPartialAsync("_ValidationScriptsPartial");} *@
```

7. **Run** the application and check the validation. The user experience should be very similar but now it is the server-side model binding system that is performing the validation.
8. **Un-comment** the reference to **_ValidationScriptsPartial.cshtml**

End of Lab

Lab 13 (Shopping Cart Part I)

Objectives

- Create a **form** used to add items to your cart
- Submit the form **asynchronously**
- Return a **message** that is inserted into the page

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

In this first part, you will add a form to the product detail page that can be used to add a number of that product to your cart. The form will submit to the server asynchronously (using Ajax and jQuery).

2. Use the **form tag helper** to add a form to the bottom of **Details.cshtml** (after the final `</div>` tag).

```
<br>
<form id="addToCartForm" asp-controller="Product"
      asp-action="AddToCart" asp-route-id="@Model.Id">

    <input name="quantity" size="3" />
    <input type="submit" value="Add to Cart" />
</form>
```

Supplier

Exotic Liquids

Edit | [Back to List](#)

Add to Cart

3. Add a new action to **ProductController** named **AddToCart**. This action should accept two parameters (**id** of type **int** and **quantity** of type **int**).

```
[HttpPost("product/addtocart")]
public async Task<IActionResult> AddToCart(int id, int quantity)
```

Notice that the action is simply accepting the two form field values as individual arguments. In this case, there is really no need to create a separate view model.

4. Add code to the **AddToCart** action that retrieves the correct **product**, calculates the **total cost** (based on quantity), creates an appropriate **message**, and passes the message to a **partial view**.

```
[HttpPost("product/addtocart/{id}")]
public async Task<IActionResult> AddToCart(int id, int quantity)
{
    var product = await _repository.GetProduct(id);
    var totalCost = quantity * product.UnitPrice;

    string message = $"You added {product.ProductName} " +
                    $"(x {quantity}) to your cart " +
                    $"at a total cost of {totalCost:C}.";

    return PartialView("_AddedToCart", message);
}
```

The next step will be to create the `_AddedToCartPartial` view which should be strongly-typed to a string. We are not recording the user's purchase on the server-side yet. We will do that in the next lab.

There is another compiler warning here for if the product is null. Like before, we will address this in the section on error handling.

5. Add a new view (Empty template) named **_AddedToCart.cshtml** to the **Views/Product** folder and add some markup to display the **message** and a link to "continue shopping".

```
@model string
<br>
<div id="message" class="alert alert-success" role="alert">@Model</div>
<p>
    <a asp-controller="Home" asp-action="Index">Continue Shopping</a>
</p>
```

6. **Run** the application, view the details for a product, and test the form. You should see the message but as a full-page reload (no menu or footer).

You added Chai (x 3) to your cart at a total cost of \$54.00.

[Continue Shopping](#)

Instead of having the partial view be the entire response, we would like to submit the form via an Ajax request and then inject the response into the existing page.

7. Add a **<div>** element to the bottom of **Details.cshtml** (beneath the form) to act as a placeholder for where we will inject the response from the server.

```
<div id="message"></div>
```

8. Add a **section** to the bottom of **Details.cshtml** named **scripts** with a reference to a JavaScript file named **cart.js**

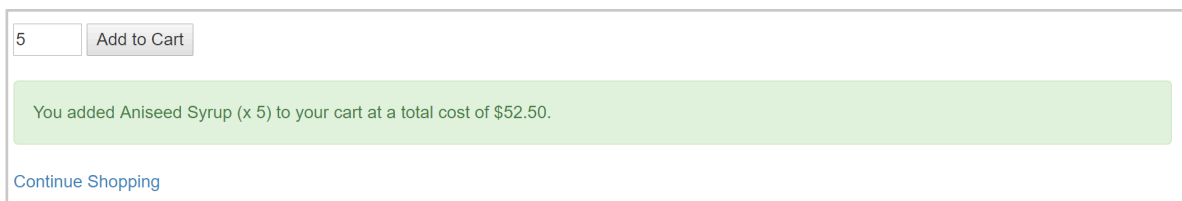
```
@section scripts {  
    <script src="~/js/cart.js" asp-append-version="true"></script>  
}
```

By using the scripts section (defined in `_Layout.cshtml`), we ensure the JavaScript reference is in the proper location on the page (immediately before the `</body>` tag).

9. Right-click on the **/wwwroot/js** folder, choose **[Add > Existing Item...]**, and select the **cart.js** file from the **Lab13** folder.

If you would like, examine the JavaScript code in `cart.js`. This code uses jQuery to post the `addToCartForm` to the server programmatically by using the jQuery ajax function.

10. **Run** the application and confirm that the add to cart operation now results in a partial-page update.



At this point, we don't actually have a shopping cart on the server-side. We also don't have a way to view the contents of the user's cart. Those things will be addressed in the next lab.

End of Lab

Lab 14 (Shopping Cart Part 2)

Objectives

- Keep track of the user's cart by using **session state**
- Implement the **view cart** page

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

*If you do not want to use your solution from the previous lab, you can open the solution in the **Begin** folder for this lab.*

2. Add code to **Program.cs** (before the call to `AddControllersWithViews`) to add **in-memory caching** and **session state**.

```
builder.Services.AddMemoryCache();  
builder.Services.AddSession();
```

3. Add the **session middleware** to the request pipeline. Add this **before** the call to **UseRouting**.

```
app.UseSession();
```

*The next step will be to define a type that will be used to represent the user's shopping cart. This is a model object but one that's specific to the web application. Therefore, we will define this type in the **EComm.Web** project.*

4. Right-click on the **Models** folder in **EComm.Web**, choose [**Add > Existing Item...**], and select the **ShoppingCart.cs** file from the **Lab 14** folder.

*This class defines the basic structure of a **ShoppingCart** type that consists of a collection of **LineItem** objects. It also defines a helper method for retrieving a **ShoppingCart** object from session and another for putting a **ShoppingCart** object into session.*

5. Take a moment to **examine** the code in **ShoppingCart.cs**

6. Add code to the **AddToCart** action in **ProductController** that updates the cart. This code needs to retrieve the cart and determine if there is already a line item for the product. If a line item already exists, increment it. If not, create a new line item.

```
string message = $"You added {product.ProductName} " +
    $"(x {quantity}) to your cart " +
    $"at a total cost of {totalCost:C}.";

var cart = ShoppingCart.GetFromSession(HttpContext.Session);
var lineItem = cart.LineItems.SingleOrDefault(item => item.Product.Id == id);
if (lineItem != null) {
    lineItem.Quantity += quantity;
}
else {
    cart.LineItems.Add(new ShoppingCart.LineItem {
        Product = product,
        Quantity = quantity
    });
}
ShoppingCart.StoreInSession(cart, HttpContext.Session);

return PartialView("_AddedToCart", message);
```

The next task is create the "view cart" page.

7. Add a new action to **ProductController** named **Cart** that passes the cart to a view.

```
[HttpGet("product/cart")]
public IActionResult Cart()
{
    var cart = ShoppingCart.GetFromSession(HttpContext.Session);
    return View(cart);
}
```

8. Right-click on the **Views/Product** folder, choose [**Add > Existing Item...**], and select **Cart.cshtml** from the **Lab14** folder.
9. Take a moment to **examine** the markup contained in **Cart.cshtml**.

The final task is to add a menu item to the top of _Layout.cshtml so the user can easily navigate to the shopping cart page.

- I0.** In **_Layout.cshtml**, add a new item that will appear on the **right side** of the navigation bar.

```
<li class="nav-item">
  <a class="nav-link text-dark" asp-area=""
    asp-controller="Home" asp-action="Privacy">Privacy</a>
</li>
<li class="nav-item">
  <a class="nav-link text-dark" asp-controller="Product"
    asp-action="Cart">View Cart</a>
</li>
```

- I1. Run** the application and test the functionality. Specifically, add some things to your cart and then view the shopping cart page.

End of Lab

If you have extra time, a bonus task would be to eliminate the hard-coded dependency on `HttpContext` in the `AddToCart` action. This dependency would make the `AddToCart` action more difficult to test since the `HttpContext` will need to be present for the action to function.

Hint: Research the `HttpContextAccessor` type and think about what should be injected into `ProductController`.

Lab 15 (Shopping Cart Part 3)

Objectives

- Add a **checkout form** to the cart page
- Implement the form as a **partial view**
- Include **input validation**

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

*If you do not want to use your solution from the previous lab, you can open the solution in the **Begin** folder for this lab.*

We will be adding a form to the existing cart page. This means that when the cart page is generated, multiple things will now need to be sent to the page (the cart object and the customer information). So, we will create a view model to provide the view with everything it needs.

2. Add a new record to the **Models** folder named **CartViewModel** with the following code.

```
public record CartViewModel
(
    ShoppingCart? Cart = null,
    [Required]
    string? CustomerName = null,
    [Required]
    [EmailAddress]
    string? Email = null,
    [Required]
    [CreditCard]
    string? CreditCard = null
);
```

3. Modify the **Cart** action in **ProductController** so that it creates a **CartViewModel** and passes it to the view.

```
[HttpGet("product/cart")]
public IActionResult Cart()
{
    var cart = ShoppingCart.GetFromSession(HttpContext.Session);
    var cvm = new CartViewModel(Cart: cart);
    return View(cvm);
}
```

4. Change the model directive in **Cart.cshtml** to **CartViewModel** and modify the **foreach** statement and **grand total** content accordingly.

```
@model CartViewModel

<h1>Shopping Cart</h1>
<div class="row">
    ...
    @if (Model.Cart != null) {
        @foreach (var lineItem in Model.Cart!.LineItems)
        {
            ...
            <td>@Model.Cart!.FormattedGrandTotal</td>
            ...
        }
    }
```

The next task is to add the HTML form for the checkout process. Since this involves a significant amount of markup, the code is included in the lab bundle as a partial view.

5. Right-click on the **Views/Product** folder, choose [**Add > Existing Item...**] and select **_CheckoutForm.cshtml** from the **Lab15** folder.
6. Take a moment to **examine** the contents of **_CheckoutForm.cshtml**. Take special note of the tag helpers related to **validation**.
7. Modify **Cart.cshtml** to that it includes the **_CheckoutForm** partial view.

```
</tbody>
</table>
<partial name="_CheckoutForm" />
</div>
</div>
```

In order for client-side validation to function, the jQuery validation JavaScript must be included. A partial view with the necessary files is already part of the project but it is not included in the layout since they will most likely not be needed on every page.

8. Add a **scripts section** to the bottom of **Cart.cshtml** that uses the **_ValidationScripts** partial view.

```
@section Scripts {  
    <partial name="_ValidationScriptsPartial" />  
}
```

9. **Run** the application, add something to your cart, and then try to checkout. Leave everything **blank** at first and check that the **client-side validation** is working.

The screenshot shows a web form titled "Checkout". At the top, there are three red bullet points indicating validation errors: "The CustomerName field is required.", "The Email field is required.", and "The CreditCard field is required.". Below these are three input fields. The first field is labeled "CustomerName" and has a red error message "The CustomerName field is required." below it. The second field is labeled "Email" and has a red error message "The Email field is required." below it. The third field is labeled "CreditCard" and has a red error message "The CreditCard field is required." below it. At the bottom of the form is a green button labeled "Checkout".

The form labels don't look quite right (e.g. CustomerName). We can fix this by adding the [Display] attribute to the properties of the view model. Feel free to do that if you have some extra time.

The next task will be to create an action that can receive the submission of the checkout form.

10. Add a **new action** to **ProductController** named **Checkout** with the **HttpPost** attribute that has a **CartViewModel** as a parameter.

```
[HttpPost("product/checkout")]  
public IActionResult Checkout(CartViewModel cvm)
```

The model binding system will automatically populate the CartViewModel from the incoming request. During that process, the model binding system will also execute server-side validation based on the data annotations that are on the view model's properties. So, it is important to check if that validation failed.

- 11.** Add code to the Checkout action that checks the **ModelState.IsValid** property. If **false**, we need to set the **Cart** property of the view model to the shopping cart (from session) and return the view to the client. If **true**, we need to complete the order process and return some sort of **thank you page**.

```
[HttpPost]
public IActionResult Checkout(CartViewModel cvm)
{
    if (!ModelState.IsValid)
    {
        cvm = cvm with {
            Cart = ShoppingCart.GetFromSession(HttpContext.Session)
        };
        return View("Cart", cvm);
    }
    // TODO: Charge the customer's card and record the order
    HttpContext.Session.Clear();
}
```

- 12.** Add a new view to the **Views/Product** folder named **ThankYou.cshtml** (Empty template - not a partial) and provide the user with an appropriate message.

```
@{
    ViewData["Title"] = "Thank You";
}
<h2>Thank You</h2>
<p>Your order has been received.</p>
```

- 13. Run** the application and test the checkout process. For a valid credit card number, you can use "4111 1111 1111 1111" (Visa) or "5555 5555 5555 4444" (MasterCard).

End of Lab

If you have some extra time, refactor the checkout process to use the post-redirect-get pattern that was mentioned during the coverage of TempData.

Lab 16

Objectives

- Examine the **default** error behavior
- Add a **controller** for centralized error handling
- Configure the **StatusCodePages** middleware
- Modify the **ExceptionHandler** middleware

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

*If you do not want to use your solution from the previous lab, you can open the solution in the **Begin** folder for this lab.*

2. **Run** the application, view a product, and change the URL to make a request for a product that **does not exist** (e.g. /product/999).

*The result is a `NullReferenceException`. If running in **Debug** mode in Visual Studio, this will activate the exception assistant. If not running in **Debug** mode, you would see the developer exception page (stack trace, etc.)*

3. Stop the application and change the **environment** to **Production** by editing **launchSettings.json** (under the Properties node).

```
"environmentVariables": {  
  "ASPNETCORE_ENVIRONMENT": "Production"  
}
```

4. **Run** the application without debugging and make another request for **/product/999**

Error.

An error occurred while processing your request.

Request ID: |a540263d-449d48bf8f97c626.

Development Mode

Swapping to **Development** environment will display more detailed information about the error that occurred.

The Development environment shouldn't be enabled for deployed applications. It can result in displaying sensitive information from exceptions to end users. For local debugging, enable the **Development** environment by setting the **ASPNETCORE_ENVIRONMENT** environment variable to **Development** and restarting the app.

This is the view returned by the Exception Handling Middleware which is setup to call the Error action of the HomeController. However, this scenario should result in a 404 and not a 500.

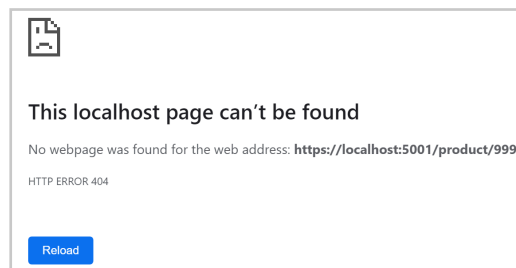
5. Stop the application and change the **environment** back to **Development**.
6. Open **ProductController.cs** and think about where we should add some **error handling** code.
7. Modify the **Details** action to check for a **null** product and return a **404**.

```
public async Task<IActionResult> Details(int id)
{
    var product = await _repository.GetProduct(id, includeSupplier: true);
    if (product is null) return NotFound();
    return View(product);
}
```

We don't have to use an exception handler to catch this situation because ECommDataContext uses SingleOrDefault which returns a null if no records are returned.

We should also think about what will happen if the database is not reachable at all. We will address that later in this lab.

8. Add a similar check in the **Edit** action and the **AddToCart** action.
9. **Run** the application again and make another request for **/product/999**



The status code is correct (404) but since the response body is blank, we are leaving it up to the browser to decide what to show the user (image above is from Google Chrome).

10. Add a new controller named **ErrorController**.

Although not necessary, it is helpful to have a dedicated controller that is responsible for the user experience in the case of an error.

11. Replace the Index action in **ErrorController** with an action for handling some common client errors.

```
[Route("clienterror")]
public IActionResult ClientError(int statusCode)
{
    ViewBag.Message = statusCode switch {
        400 => "Bad Request (400)",
        404 => "Not Found (404)",
        418 => "I'm a teapot (418)",
        _ => $"Other ({statusCode})"
    };
    return View("Error");
}
```

Note that the route used here (clienterror) will be used by the StatusCodePages middleware but will not be something the user will see in the address bar of their browser.

12. Add an empty **view** for the action named **Error.cshtml** (no model).**13.** Provide some markup for **Error.cshtml**. Feel free to do anything you would like but be sure to display **ViewBag.Message** somewhere on the page.

```
@{
    ViewData["Title"] = "Error";
}

<h1>@ViewBag.Message</h1>
```

14. Open **Program.cs** and add code to enable the **StatusCodePage middleware**.

```
app.UseHttps();
}
app.UseStatusCodePagesWithReExecute("/clienterror", "?statusCode={0}");
```

It's a good idea to configure error handling middleware early in the pipeline in case later middleware generates an error. In this case, we are adding the StatusCodePage middleware immediately after configuring the Exception Handling middleware.

- I5. Run** the application again, make another request for **/product/999**, and confirm that things are working as they should.

We will now turn our attention to handling a runtime exception. For this example, we will use the scenario of the database server being unreachable.

- I6.** To simulate a **database failure**, introduce an **error** in the **connection string** in **appsettings.json**.

```
... Initial Catalog=Broken ...
```

- I7. Run** the application and check the behavior in both **Development** and **Production** environments.

The `ExceptionHandler` middleware (or `DeveloperExceptionHandlerPage` middleware) does catch this scenario. However, we might want to extend our error handling to record this type of error and possibly send a notification.

- I8.** Open **ErrorController.cs** and add another action that will be used for runtime exceptions.

```
[Route("servererror")]  
public IActionResult ServerError()  
{  
    var exceptionFeature =  
        HttpContext.Features.Get<IExceptionHandlerPathFeature>();  
  
    var route = exceptionFeature?.Path;  
    var ex = exceptionFeature?.Error;  
  
    // TODO: write the error to a log  
    ViewBag.Message = "An unexpected error has occurred";  
  
    return View("Index");  
}
```

We will use the error information and address the missing logging code in a future lab.

We are using the same view for both error types but you could easily use a custom view for each type of error.

- 19.** Modify the **ExceptionHandler** middleware in **Program.cs** to use the new **ServerError** action.

```
app.UseExceptionHandler("/servererror");
```

- 20. Run** the application to test the exception handling behavior. Remember that you will need to specify the **Production** environment or else you will trigger the **DeveloperExceptionPage** middleware.
- 21.** When done testing, make sure the **connection string** is fixed and the environment is set back to **Development**.

End of Lab

Lab 17

Objectives

- Create a new **xUnit** test project
- Define and run a **simple test**
- Create a **stub** object
- Define and run a test for a **controller action**

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

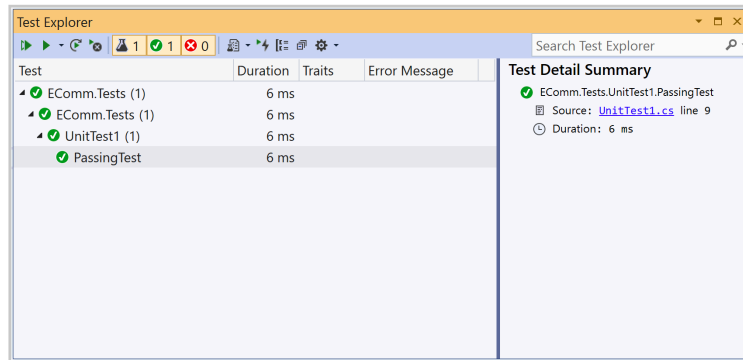
2. From the Visual Studio menu, select [**File > Add > New Project...**], choose the **xUnit Test Project** template, and name the project **EComm.Tests**

We will only be adding one unit test project in this lab. However, it is very common for a solution to contain multiple unit test projects.

3. Replace the test in **UnitTest1.cs** with a simple test that we can use to confirm that the test runner is working.

```
[Fact]
public void TwoPlusTwo()
{
    Assert.Equal(4, (2 + 2));
}
```

4. Build the solution and then open the Visual Studio Test Runner by selecting [**Test > Test Explorer**] from the Visual Studio menu. It might take a few moments before the new test appears in the list.
5. Expand the nodes until you find **PassingTest**. Right-click on the test, click **Run**, and confirm that the test **passes**.



The next objective will be to write a test for the Detail action of ProductController.

6. Right-click on the **Dependencies** node in the **EComm.Tests** project, choose [**Add Project Reference...**], and check the boxes next to **EComm.Core** and **EComm.Web**

Notice that we are not adding a reference to EComm.Infrastructure because we would like to test the ProductController independent of EF. We will need to pass an implementation of IRepository to ProductController but we should be able to pass any implementation and so we will define and use a stub.

7. Add a new test named **ProductDetails** to the **UnitTest1.cs** file.

```
[Fact]
public async Task ProductDetails()
{
    // Arrange

    // Act

    // Assert
}
```

8. Right-click on the **EComm.Tests** project, choose [**Add > Existing Item...**] and select **StubRepository.cs** from the **Lab17** folder.
9. Take a moment to **examine** the **StubRepository** class.

Notice that StubRepository implements all of the methods of IRepository but most of them throw a NotImplementedException. For now, we are only implementing the method we need to test the Details action of ProductController.

- 10.** Finish the **ProductDetails** test method. You will need to add some **using** directives.

```
// Arrange
var repository = new StubRepository();
var pc = new ProductController(repository, null);

// Act
var result = await pc.Details(1);

// Assert
Assert.IsAssignableFrom<ViewResult>(result);
var vr = result as ViewResult;
Assert.IsAssignableFrom<Product>(vr!.Model);
var model = vr.Model as Product;
Assert.Equal("Bread", model!.ProductName);
```

- 11. Build** the solution.

- 12. Run** the new test and confirm that it passes. Feel free to modify the data so that the test will fail and check the behavior.

End of Lab

If you completed the shopping cart labs and have some extra time, create unit tests for the shopping cart functionality. This will require some additional effort if you did not refactor the `HttpContext.Session` dependency (bonus task at the end of lab 14).

Lab 18

Objectives

- Create a **Web API** project
- Define an API endpoint for **retrieving** product data

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

2. From the Visual Studio menu, select [**File > Add > New Project...**], choose the **ASP.NET Core Web API** template.
 - a. Name the project **EComm.API**
 - b. Choose an appropriate **location** for the project.
 - c. Select the **None** as the Authentication Type
 - d. Ensure **Configure for HTTPS** is checked
 - e. Ensure **Enable Docker** is **not** checked
 - f. Ensure **Use controllers** is checked
 - g. Ensure **Enable OpenAPI support** is checked
3. Right-click on the **Dependencies** node in the **EComm.API** project and add project references for **EComm.Core** and **EComm.Infrastructure**
4. Copy the **connection string** from appsettings.json in EComm.Web and add it to the appsettings.json file in the **EComm.API** project.
5. Add the code to Program.cs to register as service for the **IRepository** interface (like we did earlier for EComm.Web).

```
var connStr = builder.Configuration.GetConnectionString("ECommConnection");
if (connStr == null) {
    throw new Exception("Missing connection string");
}
else {
    builder.Services.AddScoped<IRepository>(sp =>
        RepositoryFactory.Create(connStr));
}
```

6. Delete **WeatherForecastController.cs** and **WeatherForecast.cs** from the project.
7. Add a new controller named **ProductController** using the **API Controller - Empty** template in Visual Studio and remove "api" from the Route attribute.

```
[Route("[controller]")]
[ApiController]
public class ProductController : ControllerBase
{
}
```

8. Add a private field and a constructor so that we can receive an **IRepository** instance.

```
public class ProductController : ControllerBase
{
    private readonly IRepository _repository;

    public ProductController(IRepository repository)
    {
        _repository = repository;
    }
}
```

9. Add an action to ProductController that returns **all of the products** from the database (without the related suppliers).

```
[HttpGet()]
public async Task<IEnumerable<Product>> GetAllProducts()
{
    return await _repository.GetAllProducts();
}
```

10. **Run** the application and use SwaggerUI to test the action. You may want to set EComm.API as the startup project in Visual Studio.

If you get an `SqlException`, check to make sure the connection string does not still include "Initial Catalog=Broken" from the error handling lab (it should be set to EComm).

*Instead of using the Swagger UI, you could also make a request directly to **/product** with the web browser or use a tool like Postman (or curl from the PowerShell prompt).*

- 11.** Add a method to `ProductController` that returns a **single product**. Include the appropriate **`ProducesResponseType`** attributes.

```
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> GetProduct(int id)
{
    var product = await _repository.GetProduct(id);
    if (product is null) return NotFound();
    return Ok(product);
}
```

- 12.** Run the application and **test** for a product that does exist as well as for one that does not exist. Notice that a JSON body is automatically generated for both cases.

There is an overload of the `NotFound` method that accepts an argument of type object. This can be used to customize the body of the 404 response (system will serialize the provided object into JSON).

End of Lab

Lab 19

Objectives

- Add an API method to **update** a product
 - Add an API method to **create** a new product
 - Add an API method to **delete** a product
-

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

For the update operation, we will use PUT verb to replace the existing resource.

2. Add a new action to **ProductController** in the EComm.API project for **updating** a product.

```
[HttpPut("{id}")]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<IActionResult> PutProduct(int id, Product product)
{
    if (id != product.Id) return BadRequest();

    bool b = await _repository.SaveProduct(product);
    if (!b) return NotFound();

    return NoContent();
}
```

Notice that this action accepts a parameter of type Product. The model binding system is happy to populate that for us based on the JSON in the body of the incoming request. However, you could also use a different type here (DTO) based on the format you would like to accept from the client.

3. **Run** the application and test the PUT action. You can use Swagger UI and modify the request body template that it provides for the request. The supplier property should not be included in the request.

4. Add another action for **adding** a new product. This method should use the **POST** verb and include the proper attributes. The **CreatedAtAction** method should be used so the **Location** header is included in the response.

```
[HttpPost()]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> AddProduct(Product product)
{
    await _repository.AddProduct(product);
    return CreatedAtAction("GetProduct",
                           new { id = product.Id }, product);
}
```

We could try to test this code but we would then realize that we have not yet provided an implementation of the AddProduct method in the RepositoryEF class.

5. Implement the **AddProduct** method of **RepositoryEF**.

```
public async Task AddProduct(Product product)
{
    Products.Add(product);
    await SaveChangesAsync();
}
```

6. **Run** the application and **test** the POST action in a similar way to how you tested the PUT action. The product's id should not be included in the request (the id is set by the database). The supplier property should also not be included in the request but the supplierId property should be set.

```
{
  "productName": "TestProduct",
  "unitPrice": 25.00,
  "package": "Box",
  "isDiscontinued": false,
  "supplierId": 1
}
```

The response for the AddProduct action should include the Location header and should include the product in the body of the response (as JSON).

7. Add another action to **delete** a product. This method should use the **DELETE** verb and include the proper attributes. A 404 should be returned if an attempt is made to delete a product that does not exist.

```
[HttpDelete("{id}")]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<IActionResult> DeleteProduct(int id)
{
    var product = await _repository.GetProduct(id);
    if (product is null) return NotFound();
    await _repository.DeleteProduct(product);
    return NoContent();
}
```

This action retrieves the product before passing it to the repository for deletion. This is necessary since the DeleteProduct method of IRepository requires a product.

8. Implement the **DeleteProduct** method of **RepositoryEF**.

```
public async Task<bool> DeleteProduct(Product product)
{
    Products.Remove(product);
    int rowsAffected = await SaveChangesAsync();
    return (rowsAffected > 0);
}
```

9. **Run** the application and attempt to delete the product with an id of 1. You should receive a response with **exception** information in the body.

The exception happened because you attempted to delete a product that has some related orders in the database. We could enable cascading deletes in the database if that was appropriate. Another approach would be catch the exception and return a better response for the client. Try to implement one of these approaches if you have some extra time.

End of Lab

Lab 20

Objectives

- Enable cookie-based **authentication**
 - Define an **authorization policy**
 - Enforce authorization for a **controller action**
-

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

Our objective in this lab is to only allow admins to edit a product. If a non-authenticated user tries to edit a product, they should be redirected to the login page.

2. Add the authentication service to **Program.cs** in the **EComm.Web** project.

```
builder.Services.AddAuthentication(  
    CookieAuthenticationDefaults.AuthenticationScheme)  
    .AddCookie(options => { options.LoginPath = "/login"; });
```

3. Also add the **Authorization** service and define a **policy** named **AdminsOnly** that requires a role claim of **Admin**. You will need to add a **using** directive.

```
builder.Services.AddAuthorization(options => {  
    options.AddPolicy("AdminsOnly", policy =>  
        policy.RequireClaim(ClaimTypes.Role, "Admin"));  
});
```

4. Add the **authentication middleware** immediately before the authorization middleware.

```
app.UseAuthentication();  
app.UseAuthorization();
```

This takes care of the services and middleware. Now, we need to work on the UI components (e.g., login page)

5. Add a new record to the **Models** folder named **LoginViewModel** that will represent the data received from the login form.

```
public record LoginViewModel
(
    [Required]
    string Username = "",
    [Required]
    [DataType(DataType.Password)]
    string Password = "",
    string returnUrl = ""
);
```

The returnUrl will be used to return the user to the page that they were trying to access before being redirected.

6. Add a new controller named **AccountController** and replace the Index action with a login action that accepts a string for the return URL, creates a **LoginViewModel**, and passes the view model to a view.

```
public class AccountController : Controller
{
    [HttpGet("login")]
    public IActionResult Login(string returnUrl) =>
        View(new LoginViewModel(ReturnUrl: returnUrl));
}
```

7. Add a new subfolder under **Views** named **Account**.
8. Right-click on the **/Views/Account** folder, choose [**Add > Existing Item...**] and select **Login.cshtml** from the **Lab20** folder.
9. Take a moment to **examine** the contents of **Login.cshtml**

The next step is to decide which actions require authentication. In our case, we just want to secure the ability to edit a product.

10. Add an **Authorize** attribute to both **Edit** actions of **ProductController**.

```
[HttpGet("product/edit/{id}")]
[Authorize(Policy = "AdminsOnly")]
public async Task<IActionResult> Edit(int id)

[HttpPost("product/edit/{id}")]
[Authorize(Policy = "AdminsOnly")]
public async Task<IActionResult> Edit(int id, ProductEditViewModel pvm)
```

We have not written the code to handle the submission the login form yet but we can check that the redirection happens properly.

- 11. Run** the application and try to edit a product (you may need to change the startup project in Visual Studio back to EComm.Web). You should be redirected to login.

The next task is to implement the action to handle the form submission and authenticate the user.

- 12.** Add a second **Login** action to **AccountController** to handle the **post**.

```
[HttpPost("login")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel lvm)
```

- 13.** Add code to the **Login** action. The code should do the following things:

- a.** Check **ModelState** for any server-side **validation errors**.
- b.** Check the submitted **credentials**. We'll use a hard-coded username and password for now ("test" and "password").
- c.** If the credentials are **not valid**, the user should receive the login view again.
- d.** If the credentials are **valid**, create a **ClaimsPrincipal** that includes a claim for the user's name and the Admin role, call **SignInAsync**, and then redirect the user to where they were trying to go (if available).

```
public async Task<IActionResult> Login(LoginViewModel lvm)
{
    if (!ModelState.IsValid) return View(lvm);

    bool auth = (lvm.Username == "test" && lvm.Password == "password");

    if (!auth) {
        ModelState.AddModelError(string.Empty, "Invalid Login");
        return View(lvm);
    }
    var principal = new ClaimsPrincipal(
        new ClaimsIdentity(new List<Claim> {
            new Claim(ClaimTypes.Name, lvm.Username),
            new Claim(ClaimTypes.Role, "Admin")
        }, CookieAuthenticationDefaults.AuthenticationScheme));

    await HttpContext.SignInAsync(
        CookieAuthenticationDefaults.AuthenticationScheme, principal);

    if (lvm.ReturnUrl != null) return LocalRedirect(lvm.ReturnUrl);
    return RedirectToAction("Index", "Home");
}
```

*It is important that the redirect based on ReturnUrl is a **LocalRedirect**. If not, this would result in an **unvalidated redirect** which could potentially be used as part of a man-in-the-middle attack.*

- 14. Run** the application and test the functionality by trying to edit a product. You should be able to login (using "test" and "password") and be redirected back to the product.

End of Lab

If you have extra time, provide a way for an authenticated user to logout. For an additional challenge, create a page that will be displayed if the user is authenticated but not an admin or think about how you could make the Edit link only appear if the user is already authenticated as an admin.

Lab 21

Objectives

- Define a **custom** authentication scheme
- **Secure** an API method using the custom scheme

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

2. Add a new folder to the EComm.API project named **Auth** and ensure the EComm.API project is set as the **startup project** in Visual Studio.
3. Add the file **MyCustomAuthHandler.cs** from the **Lab21** folder to the Auth folder.

This class defines a constructor required by the authentication service and an override of the HandleAuthenticateAsync method that has been implemented yet.

Our custom authentication handler can look for anything in the incoming request. In this example, we will just look for the presence of a PSK in a custom header.

4. Add code to the HandleAuthenticateAsync method that checks for the presence of a header named **X-PSK** that is set to a secret value.

```
protected override Task<AuthenticateResult> HandleAuthenticateAsync()
{
    if (!Request.Headers.ContainsKey("X-PSK"))
        return Task.FromResult(AuthenticateResult.Fail("Header Not Found"));

    var psk = Request.Headers["X-PSK"].ToString();
    if (psk != "abc123")
        return Task.FromResult(AuthenticateResult.Fail("Invalid PSK"));

    // TODO: create ticket
}
```

In a production application, the code that validates the PSK will probably be much more complex (decode token, validate signature, etc.)

The next step is to create an authentication ticket with an identity and a collection of claims.

5. Replace the **TODO** comment with code that constructs a **ClaimsPrincipal** and an **AuthenticationTicket**. Return a **success** result with the ticket.

```
var principal = new ClaimsPrincipal(
    new ClaimsIdentity([
        new Claim(ClaimTypes.Name, "SomeUser"),
        new Claim(ClaimTypes.Role, "Admin")
    ], nameof(MyCustomAuthHandler)));

var ticket = new AuthenticationTicket(principal, this.Scheme.Name);
return Task.FromResult(AuthenticateResult.Success(ticket));
```

6. **Register** the authentication handler with a call to **AddAuthentication** in Program.cs.

```
builder.Services.AddAuthentication("MyCustomAuth")
    .AddScheme<AuthenticationSchemeOptions, MyCustomAuthHandler>
        ("MyCustomAuth", options => { });
```

7. Also add the **Authorization** service and define a **policy** named **AdminsOnly** that requires a role claim of **Admin**.

```
builder.Services.AddAuthorization(options => {
    options.AddPolicy("AdminsOnly", policy =>
        policy.RequireClaim(ClaimTypes.Role, "Admin"));
});
```

8. Add the authentication middleware immediately before the authorization middleware.

```
app.UseAuthentication();
app.UseAuthorization();
```

9. Add an **Authorize** attribute to the **GetProduct** API method in ProductController. Access should be restricted to **admins**.

```
[Authorize(Policy = "AdminsOnly")]
```

10. So that Swagger UI will prompt us to provide a value for the **X-PSK** header, add a parameter to the **GetProduct** method with the **FromHeader** attribute.

```
public async Task<IActionResult> GetProduct(int id,
    [FromHeader(Name = "X-PSK")] string? psk = null)
```

Adding this parameter is not necessary for our authentication to function. It simply provides an easy way to test our authentication from Swagger UI.

- I 1. Run** the application and use Swagger UI to retrieve a product with a value of **abc123** provided for **X-PSK**.
- I 2.** Try is a second time with a different value for X-PSK. You should receive a **401** response.

End of Lab

Lab 22

Objectives

- Create a new **Blazor WebAssembly** project
- Call the EComm Web API and **display products**

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for this lab.

2. Add a new project to the EComm solution named **EComm.SPA** using the **Blazor WebAssembly Standalone App** project template. Ensure that the checkbox for "Include sample pages" is checked.
3. Set **EComm.SPA** as the **startup project** and launch the application. **Explore** the three different areas by using the menu on the left side of the app.

The "Counter" page increments a variable that is maintained on the client-side. The "Weather" page retrieves data from the server when it is loaded.

4. Stop the application, open **Program.cs**, and modify the **BaseAddress** of the HttpClient to match the address of the **EComm.API** project. Your port number may be different than what is shown below. Check the launchSettings.json file in EComm.API if you need to.

```
HttpClient { BaseAddress = new Uri("https://localhost:7056")
```

5. Open **Weather.razor** for editing.

Rather than retrieving weather forecasts that are stored in a static JSON file on the server, we are going to call the EComm API.

6. Modify the `@code` section to reflect the EComm product API.

```
@code {
    private Product[]? products;

    protected override async Task OnInitializedAsync()
    {
        products = await Http.GetFromJsonAsync<Product[]>("product");
    }

    public class Product
    {
        public string ProductName { get; set; } = String.Empty;
        public decimal? UnitPrice { get; set; }
        public string Package { get; set; } = String.Empty;

        public string FormattedPrice => $"{UnitPrice:C}";
    }
}
```

Notice that the definition of Product contains less information than what will be returned by the API. This class represents what we need for display in the page. The extra fields returned by the API will simply be ignored.

7. Modify the `<h1>` and the `if` statement that checks for `null`.

```
<h1>Products</h1>

<p>This component demonstrates fetching data from the server.</p>

@if (products == null)
{
    <p><em>Loading...</em></p>
}
```

8. Modify the `HTML` to display `products` objects.

```
<thead>
    <tr>
        <th>Name</th>
        <th>Price</th>
        <th>Package</th>
    </tr>
</thead>
<tbody>
    @foreach (var product in products)
    {
        <tr>
            <td>@product.ProductName</td>
            <td>@product.FormattedPrice</td>
            <td>@product.Package</td>
        </tr>
    }
</tbody>
```

Since we will be calling an API that is on a different origin than the Blazor app (in our case, the port number is different), we need to enable CORS in EComm.Web.

9. Open **Program.cs** in EComm.API and add a **CORS** policy in the services section. Make sure to use the **correct ports** for EComm.SPA (from launchSettings.json). The ports may be different than what is shown below.

```
builder.Services.AddCors(options =>
{
    options.AddPolicy(name: "AllowedOrigins",
        builder => {
            builder.WithOrigins("http://localhost:5229",
                                "https://localhost:7207");
        });
});
```

It would be a good idea to move the list of allowed origins into app settings.

10. Add the middleware to enable **CORS** support. This should be somewhere **before** the call to **MapControllers**.

```
app.UseCors("AllowedOrigins");
```

11. **Build** the solution and confirm that you do not have any compiler errors or warnings.
12. Right-click on the EComm solution, click **Configure Startup Projects...**, and set **EComm.API** and **EComm.SPA** to both **Start without debugging**.
13. Select [**Debug > Start Without Debugging**] from the Visual Studio menu. Both applications should launch and will probably appear as separate tabs in the browser.
14. Click the **Weather** menu item in the EComm.WebApp and confirm that products are displayed.

End of Lab

If you have extra time, experiment with what happens if the Web API is not available or if the API is slow to return the data.