

# ASP.NET 3 Core Development

Visual Studio 2019

---

## About this Lab Manual

This lab manual consists of a series of hands-on lab exercises for learning to build ASP.NET Core 3 Web applications using Visual Studio 2019.

---

## System Requirements

- **.NET Core 3.1 SDK**
    - Available to download from <<https://dotnet.microsoft.com/download>>
  - **Visual Studio 2019** (any edition) version **16.4** (or later)
    - You can check the version number of Visual Studio 2019 by selecting [ Help > About Microsoft Visual Studio ]
    - Visual Studio 2019 Community Edition is available as a free download from <<https://www.visualstudio.com/>>
  - **LocalDB or SQL Server** (any version)
    - Installed by default as part of the Visual Studio installation process
    - To confirm the installation of LocalDB, 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 download an installer for LocalDB from <<http://www.microsoft.com/en-us/download/details.aspx?id=29062>> Choose the file named **SqlLocalDB.MSI**
    - If using a version of SQL Server other than LocalDB, you must be able to connect to the database with sufficient permissions to **create a new database**
  - **Postman** application for testing and debugging Web APIs
    - Available as a free download from <<https://www.getpostman.com/>>
    - A different web debugging proxy application (such as Fiddler) can be used if necessary
  - An **internet connection** is required to download and install NuGet packages from <<https://api.nuget.org>>
-

# Lab I

---

## Objectives

- Create and run a .NET Core **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 -h
```

*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 (MyFirstProject.csproj) and a source file (Program.cs).*

5. Change to the **project 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>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
</Project>
```

*Within this file, the target framework should be specified as “netcoreapp3.0”. 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.** Display the contents of the **Program.cs** file.

```
using System;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

*If you have done any C# development, nothing in this file should be new to you. There is a Main method and it prints the string "Hello World!" to the console.*

**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 any application targeting .NET Core will require.*

*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.*

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 Core console application, we’ll create a new ASP.NET Core application and see in how it is different as well as what aspects are similar.*

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 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 a Startup.cs file, two appsettings files, and a Properties directory.*

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

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
</Project>
```

*Notice that this file has the same content as the console app except for the value for the SDK (includes "Web") and the absence of the OutputType element.*

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

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

*The Main method 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 static CreateHostBuilder method uses a feature of C# known as an expression-bodied method. This is simply a shorthand way to define a method that consists of a single expression that returns a value.*

**15.** Examine the contents of **Startup.cs**. Specifically, look at where **app.UseEndpoints** is called.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
});
```

*We will talk more about this code later on. However, it looks like this code will result in the string "Hello,World!" being returned when a request is made to the root of the application ("/"). We will confirm that in a moment.*

*By default, our application is configured to listen for HTTP requests (port 5000) and HTTPS requests (port 5001). However, your machine may not be configured to trust the self-signed certificate that is used during development.*

- 16.** At the command-line, execute the CLI command to **trust the self-signed development certificate** and follow the instructions (if necessary).

```
> dotnet dev-certs https --trust
```

- 17. Run** the application and examine the **console output**.

```
> dotnet run
```

*Instead of simply printing out the string "Hello,World!" (like in the previous lab), we are told that the application is now listening on port 5000 (HTTP) and 5001 (HTTPS).*

- 18.** Open a web browser and make a request to **http://localhost:5000**. You should see the string "Hello World!"

*Notice that some logging information about the request was also printed out into the console window.*

- 19.** Make a request to **https://localhost:5001** (don't forget the "s") and confirm that you receive the same response.

- 20.** Return to the console windows and press **ctrl-c** to shut down the application.

## End of Lab

*If you have extra time, feel free to examine the other files in the project directory but will talk about those files in future sections.*

# Lab 2

---

## Objectives

- Create a new ASP.NET Core web application using **Visual Studio 2019**
- Examine the **architecture** of the application

---

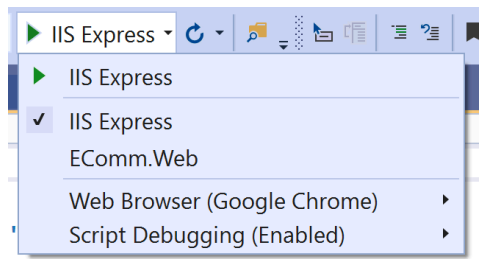
## Procedures

1. Open Visual Studio 2019, choose [ **File > New > Project...** ], select the **ASP.NET Core Web Application** and click **Next**.
  - a. Name the project **EComm.Web**
  - b. Choose an appropriate **location** for the project.
  - c. Name the solution **EComm** and click **Create**
2. In the next dialog, make the following selections:
  - a. Select **.NET Core** and **ASP.NET Core 3.1** in the drop-down boxes at the top of the dialog.
  - b. Select the **Web Application (Model-View-Controller)** template.
  - c. Ensure Enable **Docker Support** is **not checked**
  - d. Ensure **Configure for HTTPS** is **checked**
  - e. Ensure **Authentication** is set to **No Authentication**
  - f. Click **Create**
3. **Examine** the files in the solution. At this point, many of the files should look familiar to you (Program.cs, Startup.cs, appsettings.json)

*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, hosted behind IIS Express or launched as a console application (like you did in the previous lab). 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 **IIS Express** profile selected (the way it is now).
- 6. Navigate** between the two available pages (Home and Privacy).
- 7. Stop** the application and then **relaunch** it using the **EComm.Web** profile.

*Notice that with the EComm.Web profile selected, a console window appears with logging output just as it did when we used dotnet run from the command-line.*

## End of Lab

*If you have extra time, examine the additional code in Startup.cs, the HomeController class, and the Home.cshtml view. See if you can determine how an incoming request gets turned into an HTML response. This is what the instructor will do after the lab.*



# Lab 3

---

## Objectives

- Refactor the application to use **attribute-based routing**

---

## 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 modify the application to use **attribute-based routing** instead of the default dynamic route. 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. Modify the **UseEndpoints** method in ConfigureServices.
  - b. Add a **Route** attribute to each action of **HomeController** so the respective URL paths will be "/", "privacy", and "error".
  - c. Update the **UseExceptionHandler** method in ConfigureServices to use the right path for the **error page**.

3. In **ConfigureServices**, replace the call to **MapControllerRoute** with a call to **MapControllers**.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
```

4. In **HomeController.cs**, add a **Route** attribute to the **Index** action so that it acts as the home page.

```
[HttpGet("")]
public IActionResult Index()
```

5. Also add an appropriate **Route** attribute for the **Privacy** and **Error** actions.

```
[HttpGet("privacy")]
public IActionResult Privacy()
{
    return View();
}
```

```
[HttpGet("error")]
[HttpPost("error")]
[ResponseCache(...)]
public IActionResult Error()
```

*The `ExceptionHandler` middleware uses the path to the `Error` action to specify the content to return in the case of an error. We are using the `HttpGet` and `HttpPost` attributes to ensure error pages can be returned in both cases.*

*Since we just changed the route for the `Error` action, that method must also be updated.*

6. Back in **ConfigureServices**, update the call to **UseExceptionHandler** to use the new route for the **Error** action.

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

7. **Run** the application and ensure that it works as before (except the path for the privacy page is now `/privacy` instead of `/home/privacy`)

*We will use more features of the routing system as our application grows.*

## End of Lab

# Lab 4

---

## Objectives

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

---

## Procedures

*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.*

1. Open a **command prompt** (or PowerShell) and change the working directory to the **Lab04** folder (where the **EComm.sql** file is located).

2. **Execute** the following command:

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

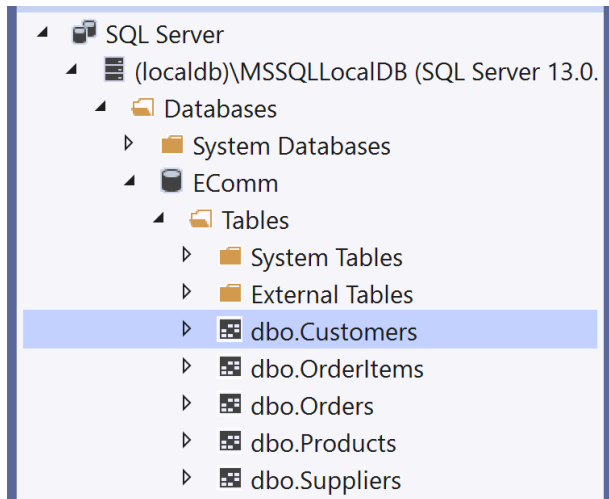
3. If not already open, re-open your **EComm** solution from the previous lab.

*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 5

---

## Objectives

- Add a **class library** for **common** data access code
- Define the **entity types**
- Define the data access **interface**
- Add a class library for the **EF data access** code
- Define an EF **DbContext** class

---

## 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 (.NET Core)** template, and name the project **EComm.Data**
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.Data** project, select [ **Add > Existing Item...** ], and add the five **.cs** files from the **Lab05** folder.
5. Take a moment to **examine** the code in each of these files. Notice that each of these types is defined in the **EComm.Data.Entities** namespace and each type corresponds to a table in the database.
6. Right-click on the **EComm.Data** project, select [ **Add > New Item...** ], select **Interface**, and name the file **IRepository.cs**
7. Change the visibility of the **Repository** to be **public**

```
public interface IRepository
```

*We are not going to be concerned at this point with trying to add every interface member that we will need. We can (and will) add more as we need them.*

8. Add methods to the interface for getting a **list of all the products** and for getting a **single product by id**. You will need to add a **using** directives for **EComm.Data.Entities** and **System.Threading.Tasks**

```
public interface IRepository
{
    Task<IEnumerable<Product>> GetAllProducts();
    Task<Product> GetProduct(int id);
}
```

*We are using a Task return type since these are IO-bound operations and we would like to be able to await on these operations.*

*The next step is to add the class library that will contain the EF-specific data access code.*

9. From the Visual Studio menu, select [ **File > Add > New Project...** ], choose the **Class Library (.NET Core)** template, and name the project **EComm.Data.EF**

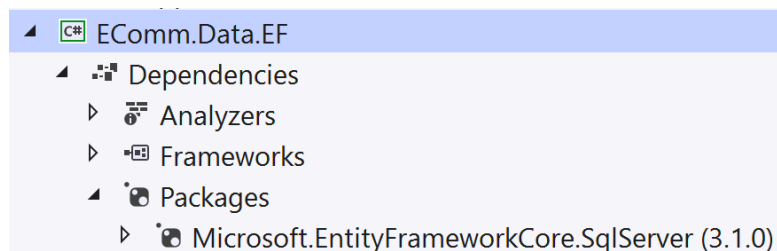
10. Delete the **Class1.cs** file.

*Before we can add any EF-specific code, we need to add a package reference to EF Core.*

11. Right-click on the EComm.Data.EF project, select [ **Manage NuGet Packages...** ] and ensure the **Browse** tab is selected.

12. Search for and install version **3.1.0** 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 correct package listed in the project under Dependencies/Packages (your version might be newer than the one shown below).*



*We will also need a reference to the EComm.Data project since EF needs to know how our entities are defined.*

- 13.** Right-click on the **Dependencies** node in the **EComm.Data.EF** project, choose **[ Add Reference... ]**, check the box next to **EComm.Data**, and click **OK**.
- 14.** Add a **new class** to the **EComm.Data.EF** project named **ECommDataContext**.
- 15.** Make the DataContext class **public** and have it inherit from **DbContext**. You will need to add a **using** directive for **Microsoft.EntityFrameworkCore** namespace.

```
public class ECommDataContext : DbContext
```

- 16.** Add a **constructor** to the **ECommDataContext** class that accepts a **DbContextOptions** object and passes it to the base class constructor.

```
public class ECommDataContext : DbContext
{
    public ECommDataContext(DbContextOptions options)
        : base(options) { }
}
```

*The DbContextOptions object will allow us to pass in the connection string as well as other provider specific options.*

- 17.** Implement the **DbSet** properties for our **entities**.

```
public class ECommDataContext : DbContext
{
    public ECommDataContext(DbContextOptions options)
        : base(options) { }

    public DbSet<Customer> Customers { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderItem> OrderItems { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Supplier> Suppliers { get; set; }
}
```

*The last thing we need to do with DbContext is make sure that DbContext implements the IRepository interface.*

**18. Add IRepository to the definition of ECommDataContext**

```
public class ECommDataContext : DbContext, IRepository
```

**19. Implement the two methods of IRepository. You will need a using directive for the System.Threading.Tasks namespace.**

```
public async Task<IEnumerable<Product>> GetAllProducts()
{
    return await Products.ToListAsync();
}

public async Task<Product> GetProduct(int id)
{
    return await Products.SingleOrDefaultAsync(p => p.Id == id);
}
```

**20. Build** the solution and address any **errors** or **warnings** that appear.**End of Lab**

*We could run the web application at this point but we won't see anything new since we are not yet using the new code.*



# 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, choose [ **Add Reference...** ], check the boxes next to **EComm.Data** and **EComm.Data.EF**, and click **OK**.

*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 would 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"  
}
```

4. Open **Startup.cs** and find the **ConfigureServices** method.
5. Add the code below to **register** our data access component as a **service**. You will need to add **using** directives for **Microsoft.EntityFrameworkCore** and **EComm.Data.EF**

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ECommDataContext>(
        options => options.UseSqlServer(
            Configuration.GetConnectionString("ECommConnection")));

    services.AddControllersWithViews();
}
```

*This code makes ECommDataContext available as a service for dependency injection. This would work but would require our controllers to accept a constructor parameter of type ECommDataContext. However, we would like our controllers to be able to use any implementation of IRepository with minimal changes to our code.*

6. Add an **additional service** so the DI system knows what to provide when an **IRepository** implementation is requested. You will need to add a **using** directive for **EComm.Data**

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ECommDataContext>(
        options => options.UseSqlServer(
            Configuration.GetConnectionString("ECommConnection")));

    services.AddScoped<IRepository, ECommDataContext>(
        sp => sp.GetService<ECommDataContext>());

    services.AddControllersWithViews();
}
```

7. Open **HomeController.cs** and add a **private field** of type **IRepository**. You will need to add a using directive for **EComm.Data**

```
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;
}
```

*When an instance of this controller is created at runtime, the DI system will pass in an instance of **ECommDataContext**. However, notice this controller only knows that it's receiving an **IRepository**. This decouples our controllers from the underlying data access layer. Using a different data access layer would only require a change to the service registration code.*

9. Modify the **Index** action of **HomeController** so that it is asynchronous (we will talk about that in the next section) and returns some model information.

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

10. So that we can view the SQL generated by EF in the log, change the log level for **Microsoft** events in **appsettings.Development.json** from **Warning** to **Information**.

```
"LogLevel": {
  "Default": "Information",
  "Microsoft": "Information",
  "Microsoft.Hosting.Lifetime": "Information"
}
```

11. **Run** the application and check the results. If a **SqlException** is thrown, double-check the accuracy of the connection string in **appsettings.json**.
12. With the application running, check the **log** for the **SQL** used by EF.

*If running with the **EComm.Web** profile, the log is displayed in the console window. If running with the **IIS Express** profile, the log is displayed in the Visual Studio Output window.*

```
info: Microsoft.EntityFrameworkCore.Database.Command[20100]
      Executing DbCommand [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 in this case since we are only interested in the count. If we using EF directly, it would be smart enough to figure that out but since we have an extra layer in between, we would need to add another method to `IRepository` or figure out a way to be more creative (which we will do later on in the course).*

- I3.** As a final experiment, modify the **Index** action to return all of the products as **JSON**.

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

- I4.** Run the application again and check the results.

*Depending on the browser you are using, the JSON data may render into the browser window (Chrome) or be treated as a file to download (IE).*

## End of Lab

# Lab 7

## Objectives

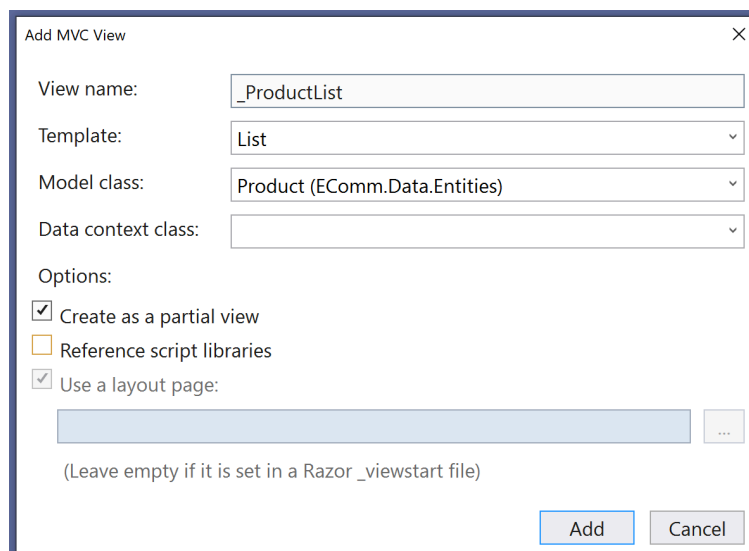
- Modify the home page to display a **list of products**

## 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...** ], make the selections as shown below, and click **Add**.



*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 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.*

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.Data.Entities.Product>
@{
    ViewData["Title"] = "Home Page";
}
```

5. Open **HomeController.cs** for editing 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	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
2	Chang	19.00	24 - 12 oz bottles	<input type="checkbox"/>	1	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
3	Aniseed Syrup	10.00	12 - 550 ml bottles	<input type="checkbox"/>	1	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
4	Chef Anton's Cajun Seasoning	22.00	48 - 6 oz jars	<input type="checkbox"/>	2	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
5	Chef Anton's Gumbo Mix	21.35	36 boxes	<input checked="" type="checkbox"/>	2	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
6	Grandma's Boysenberry Spread	25.00	12 - 8 oz jars	<input type="checkbox"/>	3	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
7	Uncle Bob's Organic Dried Pears	30.00	12 - 1 lb pkgs.	<input type="checkbox"/>	3	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

*This is the list of products from the database but there are few things that should be addressed.*

- 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.

*We will work on (a) and (b) now. The others will be addressed later on.*

7. Open **\_ProductList.cshtml** for editing 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>
```

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). We could modify our ECommDataContext to fetch the supplier for each product by using the Include method or we allow the caller to make that choice (what we will do).*

11. Modify both methods in the **IRepository** interface so that they accept an **optional parameter** that specifies whether the **suppliers** should be **included** in the results.

```
Task<IEnumerable<Product>> GetAllProducts(bool includeSuppliers = false);
Task<Product> GetProduct(int id, bool includeSupplier = false);
```

12. Modify the methods in **ECommDataContext** to accept and use the new **parameter**.

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

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

*We are using the new version of the switch statement introduced in C# 8 but a traditional if statement would work equally well here.*

- I3.** Modify the **Index** method in **HomeController** so that suppliers will be **included**.

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

- I4.** **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 also 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. You will need to add some **using** directives.

```
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**, that takes an **id** parameter, fetches the right **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...** ], and make the selections shown below:

*If the Add View... menu option does not appear, try closing Visual Studio and then reopen the solution.*

Add MVC View

View name:

Template:

Model class:

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 will automatically put the view file into 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 (that are 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** for editing 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	
<b>ProductName</b>	Aniseed Syrup
<b>UnitPrice</b>	10.00
<b>Package</b>	12 - 550 ml bottles
<b>IsDiscontinued</b>	<input type="checkbox"/>
<b>Supplier</b>	Exotic Liquids
<a href="#">Edit</a>   <a href="#">Back to List</a>	

*Unfortunately, the "Back to List" link on the Details page does not currently 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.*

- 12.** 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>
```

- 13. 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** for editing 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.chhtml** that starts with **"@model"**.
12. Open **HomeController.cs** for editing and update the **Index** action to reflect the fact that we no longer need to fetch and pass a **list of products**.

```
[Route("")]  
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.

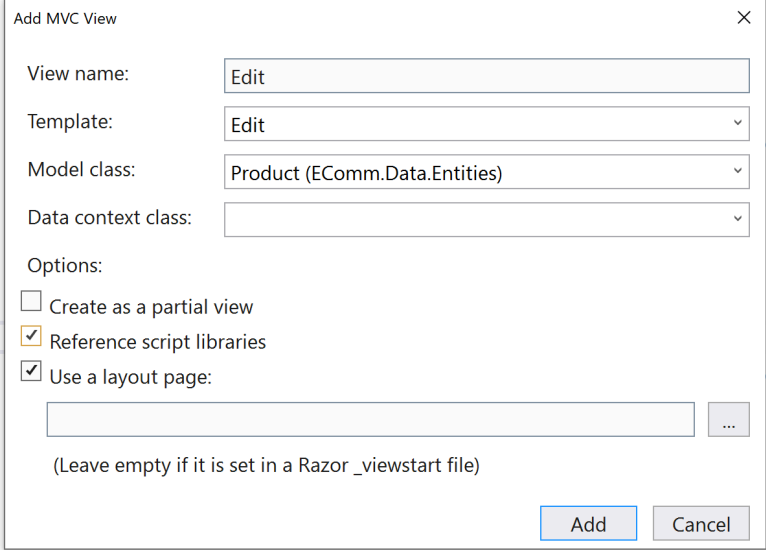
*If you do not want to use your solution from the previous lab, you can open the solution in the **Begin** folder for this lab.*

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 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 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** for editing 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...** ], and make the following selections:



View name:

Template:

Model class:

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)

*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  
1

ProductName  
Chai

UnitPrice  
18.00

Package  
10 boxes x 20 bags

☐ IsDiscontinued

SupplierId  
1

[Save](#)

[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 (five lines of markup).

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**.

We are adding this class to the `Models` folder in `EComm.Web` (and not `EComm.Data`) because it is a web-specific type that supports our view.

9. Add the **properties of a Product** to **ProductEditViewModel** plus a property for the collection of suppliers. You will need to add a **using** directive for **EComm.Data.Entities**

```
public class ProductEditViewModel
{
    public int Id { get; set; }
    public string ProductName { get; set; }
    public decimal? UnitPrice { get; set; }
    public string Package { get; set; }
    public bool IsDiscontinued { get; set; }
    public int SupplierId { get; set; }
    public Supplier Supplier { get; set; }

    public IEnumerable<Supplier> Suppliers { get; set; }
}
```

*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 the advantage of allowing us to remove or rename some of the product properties.*

*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?.Select(s => new SelectListItem
    { Text = s.CompanyName, Value = s.Id.ToString() })
    .OrderBy(item => item.Text);
```

*We are using the null-conditional operator (Suppliers?.) to prevent an exception from being thrown if this property is ever called by the system before the Suppliers property has been set.*

*We are using LINQ's data projection feature here but 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-control"></select>
</div>
```

*ProductController will fetch the list of all the suppliers but we don't have a method in IRepository for that yet.*

- 13.** Add a new method to **IRepository** that can be used to get all of the **suppliers**.

```
public interface IRepository
{
    Task<IEnumerable<Product>> GetAllProducts(...);
    Task<Product> GetProduct(...);
    Task<IEnumerable<Supplier>> GetAllSuppliers();
}
```

- 14.** Add an implementation of the **GetAllSuppliers** method to **ECommDataContext**

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

- 15.** Open **ProductController.cs** for editing 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<ActionResult> 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).*

- 16. 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 is **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 (it always will right now), **save** the modified product to the database. You will need to extend IRepository and ECommDataContext again.
  - 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. Add a new method to **IRepository** for **saving a product**.

```
Task SaveProduct(Product product);
```

### 4. Add an implementation of **SaveProduct** to **ECommDataContext**.

```
public async Task SaveProduct(Product product)
{
    Products.Attach(product);
    Entry(product).State = EntityState.Modified;
    await SaveChangesAsync();
}
```

### 5. Open **ProductController** for editing 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.Suppliers = await _repository.GetAllSuppliers();
        return View(pvm);
    }
    var product = new Product {
        Id = id,
        ProductName = pvm.ProductName,
        UnitPrice = pvm.UnitPrice,
        Package = pvm.Package,
        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 validation and error handling. We will address those issues later in the course.*

### 6. Open **Detail.cshtml** for editing and enable the **Edit** link.

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

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

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

*The only reason this is necessary is because we are using the home page for the product list page (instead of ProductController's Index action).*

8. **Run** the application and test the complete process of editing of 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**



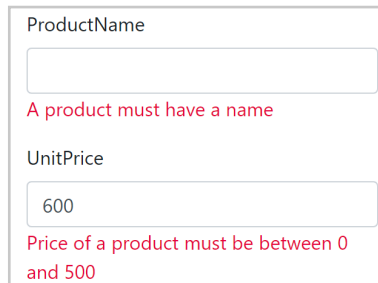
3. Open **ProductEditViewModel.cs** for editing.
4. Add the **Required** attribute to the **ProductName** property. You will need to add a **using** directive for **System.ComponentModel.DataAnnotations**

```
[Required(ErrorMessage = "A product must have a name")]  
public string ProductName { get; set; }
```

5. 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")]  
public decimal? UnitPrice { get; set; }
```

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



Product Name

A product must have a name

Unit Price

600

Price of a product must be between 0 and 500

7. Find the line in **Edit.cshtml** that reference **\_ValidationScriptsPartial.cshtml** and comment it out.

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

*This will enable you to check that the data validation is still enforced even if the client-side validation is unable to run.*

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

## End of Lab

# Lab 13

## Objectives

- Implement **shopping cart** functionality (Part I)

## 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.*

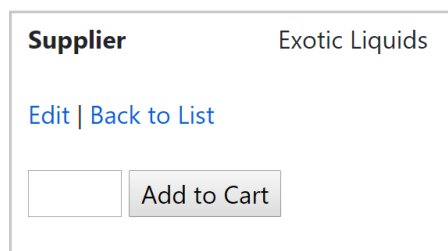
*For this lab and the next two, you will add shopping cart functionality to the EComm application.*

*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**

```
<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>
```

A screenshot of a web form for a product named 'Exotic Liquids'. The form is titled 'Supplier' and contains a text input field for 'quantity' with a size of 3. Below the input field is a button labeled 'Add to Cart'. Above the input field, there are links for 'Edit' and 'Back to List'.

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.*

*Note that we are not recording the user's purchase on the server-side yet. We will do that in the next lab.*

5. Add a new view partial 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 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.

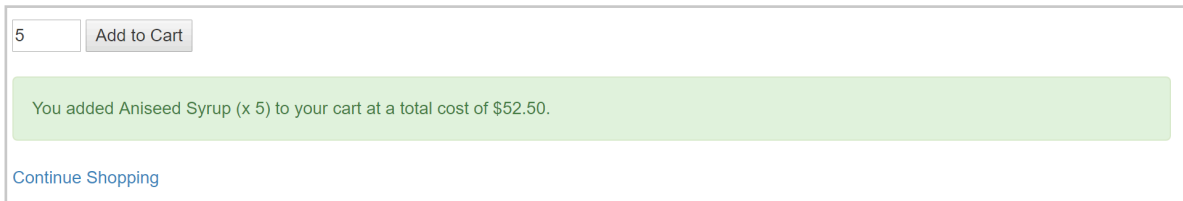
```
<div id="message"></div>
```

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 (define in `_Layout.cshtml`), we ensure the JavaScript reference is in the proper place in 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.
10. **Run** the application and confirm that the add to cart operation now results in a partial-page update.



5 Add to Cart

You added Aniseed Syrup (x 5) to your cart at a total cost of \$52.50.

[Continue Shopping](#)

*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

---

## Objectives

- Implement **shopping cart** functionality (Part 2)

---

## 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 lab, we will implement the functionality that will keep track of the products that the user has added to their cart. We will also build an action and view to provide a way for the user to view the current state of their cart.*

2. Add code to the **ConfigureServices** method in **Startup.cs** that adds **in-memory caching** and **session state**.

```
services.AddMemoryCache();  
services.AddSession();
```

3. Add code to the **Configure** method to add **session** to the request pipeline before the call to **UseEndpoints**.

```
app.UseSession();  
  
app.UseEndpoints(endpoints =>  
{  
    endpoints.MapControllers();  
});
```

*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.*

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

*Since we know that we will be storing a **ShoppingCart** object in session, we should go ahead and define some helper methods that can be used for doing that job.*

6. Add a **static method** to the **ShoppingCart** class for retrieving the cart from session.

```
public static ShoppingCart GetFromSession(ISession session)
{
    byte[] data;
    ShoppingCart cart = null;
    bool b = session.TryGetValue("ShoppingCart", out data);
    if (b) {
        string json = Encoding.UTF8.GetString(data);
        cart = JsonSerializer.Deserialize<ShoppingCart>(json);
    }
    return cart ?? new ShoppingCart();
}
```

7. Add a second **static method** for putting a cart into session.

```
public static void StoreInSession(ShoppingCart cart, ISession session)
{
    string json = JsonSerializer.Serialize(cart);
    byte[] data = Encoding.UTF8.GetBytes(json);
    session.Set("ShoppingCart", data);
}
```

8. 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.*

9. 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);
}
```

10. Right-click on the **Views/Product** folder, choose [ **Add > Existing Item...** ], and select **Cart.cshtml** from the **Lab14** folder.

11. 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.*

12. 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-area=""
        asp-controller="Product" asp-action="Cart">View Cart</a>
</li>
```

13. **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` on 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`.*

**Hint:** Research the `HttpContextAccessor` type and think about what should be injected into `ProductController` (the lab solution uses one possible solution).

# Lab 15

---

## Objectives

- Implement **shopping cart** functionality (Part 3)

---

## 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 this lab, we will add a form to the shopping cart page that allows the user to complete the checkout process.*

*The first step is to add a form to the cart page. This means that when the cart page is generated, multiple things will need to be sent to the page (the cart object and the form data). So, we will create a view model to provide that data to the view.*

2. Add a new class to the **Models** folder named **CartViewModel** with the following code. You will need to add a **using** directive.

```
public class CartViewModel
{
    public ShoppingCart Cart { get; set; }

    [Required]
    public string CustomerName { get; set; }

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

    [Required]
    [CreditCard]
    public string CreditCard { get; set; }
}
```



3. Modify the **Cart** action 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">
    ...
    @foreach (var lineItem in Model.Cart.LineItems)
    {
        ...
        <td>@Model.Cart.FormattedGrandTotal</td>
        ...
    }
```

*The next task is to add a 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.

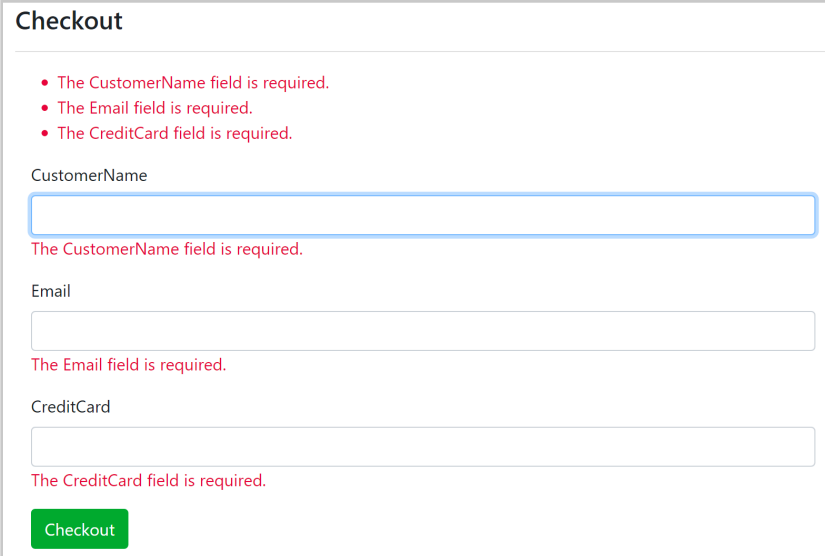
```
</tbody>
</table>
<partial name="_CheckoutForm" />
</div>
</div>
```

*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, there are three input fields. The first is labeled "CustomerName" and has a red error message "The CustomerName field is required." below it. The second is labeled "Email" and has a red error message "The Email field is required." below it. The third 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). You 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.Cart = ShoppingCart.GetFromSession(HttpContext.Session);
        return View("Cart", cvm);
    }
    // TODO: Charge the customer's card and record the order
    HttpContext.Session.Clear();
    return View("ThankYou");
}
```

- 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

- Add some **error handling**
  - Configure **status code pages**
  - Experiment with the different **environments**
- 

## 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 response is the developer exception page since an exception is thrown (`NullReferenceException`) and we are running in **Development** mode.*

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 again 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.*

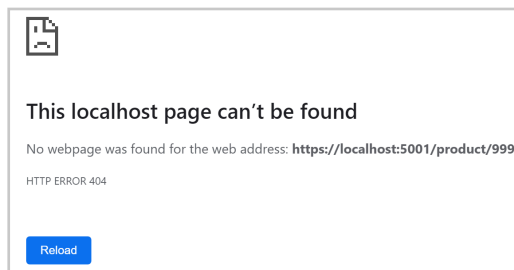
5. Stop the application and change the **environment** back to **Development**.
6. Open **ProductController.cs** for editing and think about where we should add some **error handling** code.
7. Modify the **Details** action to check for a **null** product and return a **404**.

```
public async Task<IActionResult> Details(int id)
{
    var product = await _repository.GetProduct(id, includeSupplier: true);
    if (product == 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. **Run** the application again and make another request for **/product/999**



*The status code is correct now (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 Chrome).*

9. Open **HomeController.cs** for editing and add a new **action** that will be used for **client errors**.

```
[HttpGet("clienterror")]
[HttpPost("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();
}
```

*Note that any route can be used here since the user will never see that URL in their browser.*

**10.** Add a **view** for the **ClientError** action.

**11.** Provide some markup for **ClientError.cshtml**. Feel free to do anything you would like but be sure to display **ViewBag.Message** somewhere on the page.

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

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

**12.** Open **Startup.cs** for editing and add code to enable the **StatusCodePage middleware**.

```
app.UseHsts();
}
app.UseStatusCodePagesWithReExecute("/clienterror", "{0}");

app.UseHttpsRedirection();
app.UseStaticFiles();
```

*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.*

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

- I4.** To simulate a **database failure**, introduce an **error** in the **connection string** in **appsettings.json**.

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

- I5. Run** the application and check the behavior in both **Development** and **Production** environments.

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

## End of Lab

*If you have some extra time, add some more error handling to other actions within `ProductController`.*

# 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

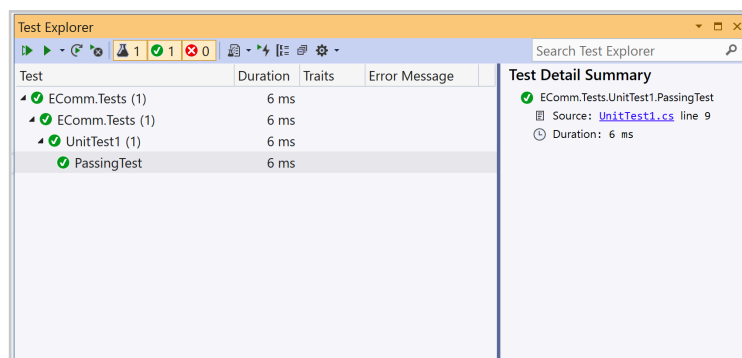
1. If not already open, re-open your **EComm** solution from the previous lab.

*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 (.NET Core)** template, and name the project **EComm.Tests**
3. Replace the test in **UnitTest1.cs** with a simple test that will confirm that the test runner is working.

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

4. Build the solution and then open the Visual Studio Test Runner by selecting [ **Test > Windows > 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 Reference...** ], and check the boxes next to **EComm.Data** and **EComm.Web**

*Notice that we are not adding a referent to EComm.Data.EF 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. So, we will define and use a stub.*

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

```
[Fact]
public void 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, null);

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

// 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 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 back in lab 14. One example is in the solution for this lab.*

# Lab 18

---

## Objectives

- Implement forms-based **authentication**
  - Add **authorization** to a controller
- 

## 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-admin attempts to edit a product, they should be redirected to the login page.*

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

3. Add the authentication service in **ConfigureServices**.

```
services.AddMemoryCache();  
services.AddSession();  
services.AddAuthentication(  
    CookieAuthenticationDefaults.AuthenticationScheme)  
    .AddCookie(options => { options.LoginPath = "/login"; });
```

4. 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.

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

5. Enable the authentication middleware in the **Configure** method.

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

*This takes care of the services and middleware. Now, we need to work on the UI and authorization rules.*

6. Add a new class to the **Models** folder named **LoginViewModel** that will represent the data received from the login form. You will need to add a **using** directive.

```
public class LoginViewModel
{
    [Required]
    public string Username { get; set; }
    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

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

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

7. Add a **new controller** named **AccountController** and define an action named **Login** that accepts a string for the return URL, creates a **LoginViewModel**, and passes the view model to the default view for the action. You will need a **using** directive.

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

8. Add a new subfolder under **Views** named **Account**.
9. Right-click on the **/Views/Account** folder, choose [ **Add > Existing Item...** ] and select **Login.cshtml** from the **Lab18** folder.
10. 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 add a product.*

11. Add an **Authorize** attribute to both **Edit** actions of **ProductController**. You will need a **using** directive.

```
[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.*

- 12. Run** the application and try to edit a product. You should be redirected to the login page.

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

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

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

- 14. Add** code to the **Login** action. The code should do the following things (try to do this on your own before looking at the code on the next page):
- 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 very 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.*

- 15. 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, think about how you could only make the Edit link appear on the list if the user is an admin (one possible approach is in the solution for this lab).*

# Lab 19

## Objectives

- Build a Web API 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. Add a new top-level **folder** to the EComm.Web project named **API**.
3. Right-click on the API folder, choose **[ Add > Controller... ]**, select the **API Controller - Empty** template, and name the controller **ProductApiController**.

```
[Route("api/[controller]")]
[ApiController]
public class ProductApiController : ControllerBase
```

*We could name this controller ProductController since it's in a different namespace from our existing ProductController but that might end up being a bit confusing.*

4. Delete the **Route** attribute from ProductApiController.

*If we left the Route attribute in place, all actions within in this controller would have a route that starts with /api/productapi/. In some applications, this might be okay but we will customize the route for each action.*

5. Add the necessary code to receive an **IRepository** and **ILogger** via DI (similar to our other controllers).

```
[ApiController]
public class ProductApiController : ControllerBase
{
    private readonly IRepository _repository;
    private readonly ILogger<ProductApiController> _logger;

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

6. Add an **action** to return **all of the products**. You should try to create this action on your own and then compare it to the code below.

```
[HttpGet("api/products")]
[ProducesResponseType(StatusCodes.Status200OK)]
public async Task<ActionResult<List<Product>>> GetProducts()
{
    var products = await _repository.GetAllProducts(includeSuppliers: true);
    return products.ToList();
}
```

7. Add another **action** to return a **single product**. Once again, you should try to create this action on your own and then compare it to the code below.

```
[HttpGet("api/product/{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<ActionResult<Product>> GetProduct(int id)
{
    var product = await _repository.GetProduct(id, true);
    if (product == null) return NotFound();
    return product;
}
```

8. **Run** the application and test both actions using the web browser.
9. Make a request for **/api/product/999** and examine the response.

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.4",
  "title": "Not Found",
  "status": 404,
  "traceId": "|75cf8f09-44bf45896942cbcf."
}
```

*We are getting the correct status code (404) and we are receiving JSON instead of HTML (from the StatusCodePages middleware). This is because the StatusCodePages middleware only triggers when there is an empty content body. Since we are using the ApiController attribute, the system is generating this content body for us. If you would like, comment-out [ApiController] and try it again.*

10. With the application running, open the **Postman** application.
11. So that you can view responses sent via HTTPS, from the Postman menu, select [ **File > Settings** ] and disable **SSL certificate verification**.

SSL certificate verification  OFF



- 12.** Use Postman to make a GET request for **all of the products** as well as **one individual product** and explore the response data.

## **End of Lab**

# Lab 20

## Objectives

- Build a Web API for **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. Add a **PutProduct** action to ProductApiController.

```
[HttpPut("api/product/{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();

    await _repository.SaveProduct(product);
    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 open **Postman**.
4. Issue a **GET** request for a product and copy the JSON of the response.
5. Change the verb of the request to **PUT** and make sure to add a request header for **Content-Type**.

▼ Headers (1)	
KEY	VALUE
<input checked="" type="checkbox"/> Content-Type	application/json

6. Paste the JSON content that you copied into the **Body** of the request, make a **change** to the data (e.g. increase the UnitPrice), and **send** the request. If successful, you should receive a 204 response.
7. Re-issue a **GET** request for the product and confirm that the change was **saved**.

### End of Lab

*If you have extra time, try to implement create and delete for a product. This will require extending IRepository and ECommDataContext.*

# Lab 21

## Objectives

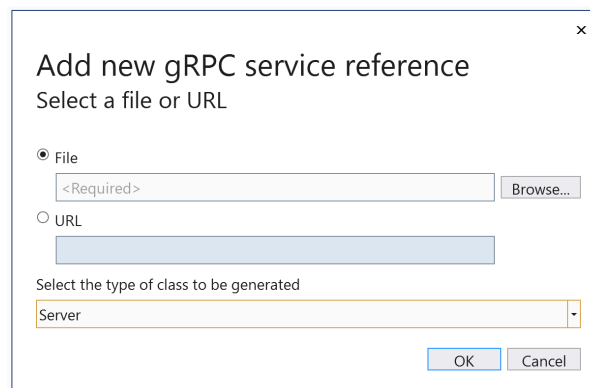
- Build a **gRPC** server and client for getting **product** information

## 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. Right-click on the EComm.Web project, choose [ **Add > Service Reference...** ], and click on **Add new gRPC service reference**.



3. For **File**, click browse, and select the **product.proto** file in the Lab21 directory. Ensure **Server** is selected in the drop-down and click **OK**.

*The wizard will add the proto file to the project, add the necessary NuGet packages, and edit the project file.*

4. **Build** the solution.
5. Right-click on the **API** folder in EComm.Web and add a new class named **ECommGrpcService** that is in the **EComm.Web.API.gRPC** namespace and inherits from **ECommGrpc.ECommGrpcBase**

```
namespace EComm.Web.API.gRPC
{
    public class ECommGrpcService : ECommGrpc.ECommGrpcBase
    {
    }
}
```

*If you look at the names in the proto file, you should be able to figure out where the namespace and base class come from.*

6. Add the code necessary to use **DI** (as you have done before).

```
public class ECommGrpcService : ECommGrpc.ECommGrpcBase
{
    private readonly IRepository _repository;
    private readonly ILogger<ECommGrpcService> _logger;

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

7. Add an **override** for the **GetProduct** method. Visual Studio can help with this if you start by typing in the word **override** but you will need to add the **async** keyword.

```
public override async Task<ProductReply> GetProduct(
    ProductRequest request, ServerCallContext context)
{
    return base.GetProduct(request, context);
}
```

8. Replace the implementation of the method with code that retrieves and returns the **correct product data**.

```
public override async Task<ProductReply> GetProduct(
    ProductRequest request, ServerCallContext context)
{
    var product = await _repository.GetProduct(request.Id, true);

    var reply = new ProductReply {
        Id = product.Id,
        Name = product.ProductName,
        Price = (double)product.UnitPrice.Value,
        Supplier = product.Supplier.CompanyName
    };
    return reply;
}
```

9. Open **Startup.cs** for editing and add the **Grpc** service in **ConfigureServices**.

```
services.AddGrpc();
```

10. In **Configure**, add an **endpoint** for our gRPC service.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
    endpoints.MapGrpcService<ECommGrpcService>();
});
```

*The URL of the service is determined by the package name and service name. So, our method will be at `/ecommpkg/ecommgrpc/getproduct`. To change this, you would need to change the package and/or service name in the proto file. This behavior is defined by the gRPC standard.*

*Since gRPC uses a binary protocol (by default), we will need to create a gRPC client to call our service.*

- I 1.** From the Visual Studio menu, select [ **File > Add > New Project...** ], choose the **Console App (.NET Core)** template, and name the project **EComm.ClientApp**
- I 2.** Right-click on the EComm.ClientApp project, choose [ **Add > Service Reference...** ], and click on **Add new gRPC service reference**.
- I 3.** For **File**, click browse, and select the **product.proto** file in the Lab21 directory. Ensure **Client** is selected in the drop-down and click **OK**.
- I 4.** **Build** the project.
- I 5.** Open the **Program.cs** file in **EComm.ClientApp** for editing.
- I 6.** Add code to the **Main** method to create a gRPC **channel** and **client**. You will need to add some **using** directives. Also change the **Main** method to be **async**.

```
static async Task Main(string[] args)
{
    var channel = GrpcChannel.ForAddress("https://localhost:5001");
    var client = new ECommGrpc.ECommGrpcClient(channel);
}
```

*The channel represents a long-lived connection to an RPC service. Multiple clients for different services can share the same channel (if the services are on the same host).*

- I 7.** Immediately after the client is created, add code to **call** the service and **output** the product data received.

```
var reply = await client.GetProductAsync(new ProductRequest { Id = 5 });

Console.WriteLine($"{reply.Id}, {reply.Name}, {reply.Price}, " +
    $"{reply.Package}, {reply.Supplier}");
```

- I 8.** **Build** the project.
- I 9.** **Run** the **EComm.Web** project.
- I 20.** Open a **command-prompt** and change the working directory to the folder where the EComm.ClientApp **csproj** file is.

- 21.** Execute the **dotnet run** command to run EComm.ClientApp and check the data returned.

### End of Lab

*If you have extra time, feel free to add more methods to the gRPC service. This will require additions to the proto file. The necessary code will be regenerated when each project is built - do not re-run the Add Service Reference wizard.*