Web API Development with ASP.NET Core 6

Labs

https://www.accelebrate.com
(877) 849-1850 ❖ info@accelebrate.com

Customized Technical Training

Accelebrate
ACCELERATED LEARNING, CELEBRATED RESULTS®

On-Site, Customized Private Training

Don't settle for a one-size-fits-all class! Let Accelebrate tailor a private class to your group's goals and experience. Classes can be delivered at your site or online (or a combination of both) worldwide. Visit our web site at **https://www.accelebrate.com** and contact us at **sales@accelebrate.com** for details.

Public Online Training

Need to train just 1-3 people? Attend one of our regularly scheduled, live, instructor-led public online classes. Class sizes are small (typically 3-6 attendees) and you receive just as much hands-on time and individual attention from your instructor as our private classes. For course dates, times, outlines, pricing, and registration, visit **https://www.accelebrate.com/public-training-schedule.**

Newsletter

Want to find out about our latest class offerings? Subscribe to our newsletter **https://www.accelebrate.com/newsletter**.

Blog

Get insights and tutorials from our instructors and staff! Visit our blog, **https://www.accelebrate.com/blog** and join the discussion threads and get feedback from our instructors!

Learning Resources

Get access to learning guides, tutorials, and past issues of our newsletter at the Accelebrate library, **https://www.accelebrate.com/library**.

Call us for a training quote!

877 849 1850

Accelebrate, Inc. was founded in 2002 with the goal of delivering **private training** that rapidly achieves participants' goals. Each year, our experienced instructors deliver hundreds of classes online and at client sites all over the US, Canada, and abroad. We pride ourselves on our **instructors' real-world experience** and ability to **adapt the training** to your team and their objectives. We offer a wide range of topics, including:

- Angular, React, and Vue
- JavaScript
- Data Science using R, Python, & Julia
- Excel Power Query & Power BI
- Tableau
- .NET & VBA programming
- SharePoint & Microsoft 365
- DevOps
- iOS & Android Development
- PostgreSQL, Oracle, and SQL Server
- Java, Spring, and Groovy
- Agile
- Web/Application Server Admin
- HTML5 & Mobile Web Development
- AWS & Azure
- Adobe & Articulate software
- Docker, Kubernetes, Ansible, & Git
- IT leadership & Project Management
- AND MORE *(see back)*

"It's not often that everything goes according to plan and you feel you really got full value for money spent, but in this case, I feel the investment in the Articulate training has already paid off in terms of employee confidence and readiness."

— Paul, St John's University

Like us on Facebook + Follow us on Twitter + Watch us on YouTube

Visit our website for a complete list of courses!

Adobe & Articulate
Adobe Captivate
Adobe Presenter
Articulate Storyline / Studio
Camtasia
RoboHelp

AWS, Azure, & Cloud
AWS
Azure
Cloud Computing
Google Cloud
OpenStack

Big Data
Alteryx
Apache Spark
Teradata
Snowflake SQL

Data Science and RPA
Blue Prism
Django
Julia
Machine Learning
MATLAB
Python
R Programming
Tableau
UiPath

Database & Reporting
BusinessObjects
Crystal Reports
Excel Power Query
MongoDB
MySQL
NoSQL Databases
Oracle
Oracle APEX
Power BI
PivotTable and PowerPivot
PostgreSQL
SQL Server
Vertica Architecture & SQL

DevOps, CI/CD & Agile
Agile
Ansible
Chef
Diversity, Equity, Inclusion
Docker
Git
Gradle Build System
Jenkins
Jira & Confluence
Kubernetes
Linux
Microservices
Red Hat
Software Design

Java
Apache Maven
Apache Tomcat
Groovy and Grails
Hibernate
Java & Web App Security
JavaFX
JBoss
Oracle WebLogic
Scala
Selenium & Cucumber
Spring Boot
Spring Framework

JS, HTML5, & Mobile
Angular
Apache Cordova
CSS
D3.js
HTML5
iOS/Swift Development
JavaScript
MEAN Stack
Mobile Web Development
Node.js & Express
React & Redux
Svelte
Swift
Xamarin
Vue

Microsoft & .NET
.NET Core
ASP.NET
Azure DevOps
C#
Design Patterns
Entity Framework Core
IIS
Microsoft Dynamics CRM
Microsoft Exchange Server
Microsoft 365
Microsoft Power Platform
Microsoft Project
Microsoft SQL Server
Microsoft System Center
Microsoft Windows Server
PowerPivot
PowerShell
VBA
Visual C++/CLI
Web API

Other
C++
Go Programming
IT Leadership
ITIL
Project Management
Regular Expressions
Ruby on Rails
Rust
Salesforce
XML

Security
.NET Web App Security
C and C++ Secure Coding
C# & Web App Security
Linux Security Admin
Python Security
Secure Coding for Web Dev
Spring Security

SharePoint
Power Automate & Flow
SharePoint Administrator
SharePoint Developer
SharePoint End User
SharePoint Online
SharePoint Site Owner

SQL Server
Azure SQL Data Warehouse
Business Intelligence
Performance Tuning
SQL Server Administration
SQL Server Development
SSAS, SSIS, SSRS
Transact-SQL

Teleconferencing Tools
Adobe Connect
GoToMeeting
Microsoft Teams
WebEx
Zoom

Web/Application Server
Apache httpd
Apache Tomcat
IIS
JBoss
Nginx
Oracle WebLogic

*Visit **www.accelebrate.com/newsletter** to sign up and receive our **newsletters** with information about **new courses**, **free webinars**, **tutorials**, and **blog articles**.*

Call us for a training quote! 877 849 1850 (US/Canada) or +1 678 648 3133

Web APIs with ASP.NET Core 6

Visual Studio 2022

About this Lab Manual

This lab manual consists of a series of hands-on lab exercises for learning to build Web APIs with ASP.NET Core 6 and Visual Studio 2022.

System Requirements

- **Visual Studio 2022** (any edition)
  - You can check the update version number of Visual Studio 2022 by selecting [Help > About Microsoft Visual Studio] from within Visual Studio
  - Visual Studio 2022 Community Edition is available as a free download from <https://www.visualstudio.com/>
- **.NET 6 SDK**
  - Should already be installed as part of the Visual Studio 2022 installation
  - You can check the versions of the SDK that you have installed by executing **dotnet --list-sdks** at a command prompt
  - You should have at least one SDK with a version number that starts with 6
  - You can download the SDK separately from <https://dotnet.microsoft.com/download>
- **LocalDB or SQL Server** (any version)
  - Installed by default as part of the Visual Studio installation process
  - To check if LocalDB is installed, you can execute **sqllocaldb i** from a command prompt. This should list the LocalDB instances that are available. If the command is not recognized then LocalDB is not installed.
  - If not installed, you can install LocalDB by using the Visual Studio installer via [Tools > Get Tools and Features...] from the Visual Studio menu. LocalDB is listed under "Individual Components" as "SQL Server Express 2016 LocalDB".
  - If using a version of SQL Server other than LocalDB, you must be able to connect to the server with sufficient permissions to create a new database
- An **internet connection** is required to download and install NuGet packages from <https://api.nuget.org>

Lab 1

Objectives

- Create and run a .NET 6 **console** application using the **CLI**
- Create and run an **ASP.NET Core** application using the **CLI**

Procedures

1. Open a **command prompt** window (or Windows PowerShell).

2. Change the **current directory** to a location where you would like to create some new projects.

3. Use the **dotnet** command to display a list of available **templates** for creating a new project.

```
> dotnet new
```

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

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

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

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

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

PowerShell: `> cat ConsoleApp.csproj`

Command Prompt: `> type ConsoleApp.csproj`

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```


Within this file, the target framework should be specified as "net6.0". This is referred to as a Target Framework Moniker (TFM) and specifies the framework that this app is targeting. It is possible to specify more than one TFM to build for multiple targets.

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

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

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

If you have C# development in previous versions of .NET, it may seem like there are several things missing in this file (namespace, class definition, etc.) but this file takes advantage of some new features in C# 10 to reduce the amount of code required. We will talk about these features shortly.

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

```
> dotnet restore
```

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

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

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

```
> dotnet build
```

Notice in the output of the build command that the result is a dll file (not an exe file). This is because the dotnet command is used to actually run the application. This is a portable application and this same dll file can be used to run the application on a different platform by using the dotnet command on that platform. A self-contained exe for Windows is also generated in .NET 6.

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 6 console application, we'll create a new ASP.NET Core 6 application and see what is different and what things are the same.*

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

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

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

> *The Empty template includes the minimum amount of code necessary for a functional ASP.NET Core 6 web application (i.e. Hello World). For future projects, we will use one of the other more full-featured templates.*

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

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

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

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

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

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

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

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

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

15. Examine the contents of **launchSettings.json** in the **Properties** directory and look for the **applicationUrl** setting in the **WebApp** section.

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

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

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

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

```
> dotnet run
```

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

18. Open a **web browser** and make a request to the non-secure **http** URL shown in the log. You should see the string "Hello World!"

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

19. Make a request to the secure **https** URL and confirm that you receive the same response. If the browser gives you a warning about the certificate, it is because the development certificate is not trusted by the browser (see step 16 in this lab).

> *If using Firefox as your web browser, an additional step may be required for Firefox to trust the development certificate. See https://support.mozilla.org/en-US/kb/setting-certificate-authorities-firefox for more information.*

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

End of Lab

Lab 2

Objectives

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

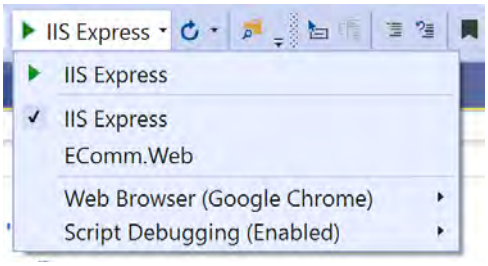
Procedures

1. Open Visual Studio 2022, choose **[File > New > Project...]**, select **ASP.NET Core Web API**, and click **Next**.
   a. Name the project **EComm.Web**
   b. Choose an appropriate **location** for the project.
   c. Change the solution name to be **EComm** and click **Next**
2. In the next dialog, make the following selections:
   a. Select **.NET 6.0** as the Target Framework
   b. Select the **None** as the Authentication Type
   c. Ensure **Configure for HTTPS** is checked
   d. Ensure **Enable Docker** is **not** checked
   e. Ensure **Use controllers** is checked
   f. Ensure **Enable OpenAPI support** is checked
   g. Click **Create**
3. **Examine** the files in the solution. The project template includes a sample controller that makes available some random weather forecast data.

> *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 **EComm.API** profile selected. A page generated by the SwaggerUI middleware should appear in the browser.

6. **Expand** the element for the **GET** endpoint, click **Try It Out**, and then click **Execute** to test the endpoint. You should see the response in the **Server response** section.

7. **Close** the browser to **stop** the application.

End of Lab

Lab 3

Objectives

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

Procedures

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

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

2. **Execute** the following command:

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

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

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

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

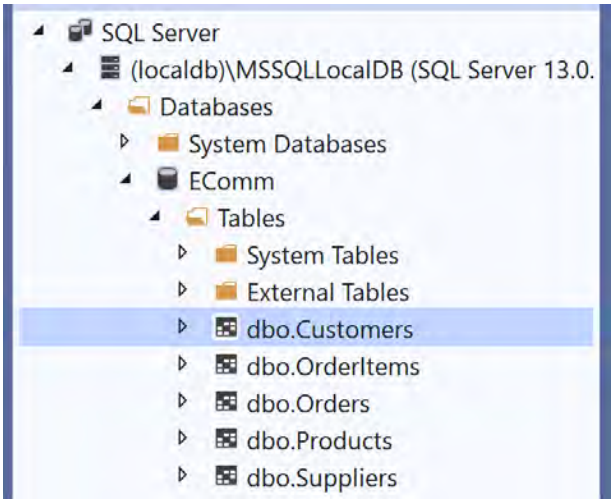
4. From the Visual Studio menu, select [**View > SQL Server Object Explorer**] and expand the **SQL Server** node.

5. If you already have an entry for **(localdb)\MSSQLLocalDB**, skip to **step 6**.

   a. To add LocalDB, right-click on the SQL Server node and select **Add SQL Server...**

   b. Expand the **Local** node, select **MSSQLLocalDB**, and click **Connect.**

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



7. Right-click on the **Products** table of the new **EComm** database and select **View Data**. You should see a collection of product records.

8. Take a moment to **explore** the rest of the database tables that are present. There are also two **stored procedures** in the **Programability** folder.

End of Lab

Lab 4

Objectives

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

Procedures

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

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

2. From the Visual Studio menu, select **[File > Add > New Project...]**, choose the **Class Library** template, name the project **EComm.Core**, and select **.NET 6.0**

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

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

4. Right-click on the **EComm.Core** project, select **[Add > Existing Item...]**, and add the six **.cs** files from the **Lab04** folder.

5. Take a moment to **examine** the code for each of the entity types (all but Repository). Notice that each of these types is defined in the **EComm.Core.Entities** namespace and each type corresponds to a table in the database.

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

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

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

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

7. Right-click on the **Dependencies** node in the **EComm.API** project, choose **[Add Project Reference...]**, check the box next to **EComm.Core**, and click **OK**.

8. Add another .NET 6.0 class library project named **EComm.Infrastructure** and delete Class1.cs

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

9. Add a project reference to EComm.Infrastructure so that it references the **EComm.Core** project.

10. Add a **new class** to the EComm.Infrastructure project named RepositoryEF that implements the **IRepository** interface.

```
namespace EComm.Infrastructure
{
  internal class RepositoryEF : IRepository
  {
  }
}
```

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

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

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

End of Lab

Lab 5

Objectives

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

Procedures

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

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

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

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

3. Search for and install the latest stable version of **Microsoft.EntityFrameworkCore.SqlServer**. Make sure you select the package for EntityFrameworkCore and not EntityFramework (the one for .NET Framework).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

> *The code above uses a switch expression (introduced in C# 8) but a simple if statement could have be used instead.*

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

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


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

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

End of Lab

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

Lab 6

Objectives

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

Procedures

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

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

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

2. Right-click on the **Dependencies** node in the **EComm.API** project and add project references for **EComm.Infrastructure**

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

3. Open the **appsettings.json** for editing and add an entry for the connection string.

   a. Make sure to add a **comma** at the end of the line for "AllowedHosts".

   b. The connection string below is shown on two lines but it must all be on **one line** in the JSON file.

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

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

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

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

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

5. Open **Program.cs** in EComm.API and find where the services are added.

6. Add code to register **IRepository** as a service with our factory method. You will need to add some using directives.

```
builder.Services.AddScoped<IRepository>(sp =>  RepositoryFactory.Create(
  builder.Configuration.GetConnectionString("ECommConnection")));
```

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

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

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

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

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

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

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

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

10. **Run** the application and use SwaggerUI to test the action. You can also make a request directly to **/product** with the web browser or use a tool like Postman (or curl from the PowerShell prompt).

11. Stop the application and add code to the **OnConfiguring** method of **RepositoryEF** to enable some **simple logging**.

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

12. **Run** the application again and check the **console** after executing the action. There is a lot of extra information but you should be able to find the SQL that was sent to the database.

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

13. If you would like, you can now delete **WeatherForcastController.cs** and **WeatherForecast.cs**

End of Lab

Lab 7

Objectives

- Add an API method for retrieving an **individual 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 method to ProductController that returns a **single product**. Include the appropriate **PorducesResponseType** attributes.

```
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<Product> GetProduct(int id)
{
  return await _repository.GetProduct(id);
}
```

   > *The possible 200 status code makes sense. The 404 status code is if the client makes a request for a product that does not exist (we will need to add code for that). The 400 status code is if the request is malformed (e.g., id value is missing or is not an integer).*

3. **Run** the application and perform a couple of tests. Make the requests directly with a web browser or a tool like Postman. Swagger UI will prevent you from doing some things that are invalid (send a non-integer value for id).

   a. Make a proper request for an existing product (**/product/3**). Confirm that the JSON for that product is returned and the status code is 200.

   b. Make a proper request for a product that does not exist (**/product/999**). You should see that you receive a 204 (no content) response with an empty body. This is not what we want. The response should be a 404 (resource not found).

   c. Make an improper request by sending a non-integer value for the id (**/product/three**). You should receive a 400 (bad request) response with a some JSON that describes the problem.

> *The 400 status code was generated automatically for us in test (c) because ProductController includes the [ApiController] attribute. Without that attribute, the response would be a 204 with an empty body.*

4. Modify the GetProduct action so that it returns a **404** when there is no product for the requested id. You will need to change the return type of the action.

```
public async Task<IActionResult> GetProduct(int id)
{
  var product = await _repository.GetProduct(id);
  if (product == null) return NotFound();

  return Ok(product);
}
```

5. **Run** the application and repeat the **test** for a product that does not exist. Notice that a JSON body is automatically generated.

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

End of Lab

Lab 8

Objectives

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

Procedures

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

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

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

2. Add a new action to ProductApiController for **updating** a product.

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

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

  return NoContent();
}
```

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

> *We are calling the SaveProduct method but this method has not been implemented yet in RepositoryEF. We will do that now.*

3. Implement the **SaveProduct** method of RepositoryEF. You will need to add the **async** keyword to the method signature.

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

> *By using this approach, we will only execute one database query. However, all of the properties of the incoming product need to be set and all of the columns in the database will be updated. The is appropriate here since this method is being called as the result of a PUT request. We might want to add another method to IRepository that uses a different approach for a PATCH type of operation.*

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

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

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

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

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

> *We are not including any code for error handling or logging. For example, if the database is not currently reachable. This is something we will address in a future lab.*

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

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

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

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

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

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

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

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

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

The exception happened because you attempted to delete a product that has some related orders in the database. We could enable cascading deletes in the database if that was appropriate. Another approach would be catch the exception and return a better response for the client. We will do this in a future lab.

End of Lab

Lab 9

Objectives

- Add **input 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.*

   > *When editing a product, there are some additional validation conditions that we would like to enforce. As we saw earlier, the system is already handling the case of an input type mismatch (e.g. passing a string for the id).*

2. Add a **Required** attribute to the **ProductName** property of the Product class. You will need to a using directive for System.ComponentModel.DataAnnotations.

```
[Required]
public string ProductName { get; set; } = String.Empty;
```

   > *Remember, instead of adding this attribute directly on our entity, we could create a DTO type for use by our controller and then apply the attribute to the DTO class.*

3. Use attributes to define a validation rule that specifies that a product's **unit price** is **required** and should be **between 1 and 500**.

```
[Required]
[Range(1.0, 500.0)]
public decimal? UnitPrice { get; set; }
```

4. **Run** the application and attempt to edit a product with a value that violates one of the validation rules. For example, try to set the unit price of a product to 5000 and check the response. You should receive a 400 status code with information about the error in the body.

   > *If you have extra time, add some additional validation or see if you can use the Range attribute to customize the error message returned to the client.*

End of Lab

Lab 10

Objectives

- Add a **controller** for centralized error handling
- Use the **ExceptionHandler** middleware
- Use **IExceptionHandlerPathFeature** to provide a custom response

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.*

   > *When editing a product, there are some additional validation conditions that we would like to enforce. As we saw earlier, the system is already handling the case of an input type mismatch (e.g. passing a string for the id).*

2. Add a new API controller to the EComm.API project named **ErrorController**.

```
namespace EComm.API.Controllers
{
  [ApiController]
  public class ErrorController : ControllerBase
  {
  }
}
```

3. Add a class to ErrorController.cs named **Problem** to represent the information we will return to the client in the case of a server error. This class can be public or it could be defined as a private nested class.

```
public class Problem
{
  public string Description { get; set; } = String.Empty;
}
```

   > *It would be okay to include additional information in the Problem class but we want to be sure not to reveal anything sensitive.*

4. Add an action to ErrorController named **SeverError** that returns a Problem object.

```
[Route("servererror")]
[ApiExplorerSettings(IgnoreApi = true)]
public Problem ServerError() =>
  new Problem { Description = "An unexpected error has occurred" };
```

> *Notice the use of the ApiExplorerSettings attribute since this action is not part of our public API and we do not want it to appear in tools like Swagger UI.*

5. Add the **exception handling middleware** to our application's pipeline and configure it to use the newly added ServerError action. Since this middleware is related to error handling, it is a good practice to put it early in the pipeline.

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

6. **Run** the application and try to **delete** the product with an id of 1 again. Last time, this returned a response with details about the exception (including the stack trace) in the body. This time, the status code should be a 500 but the body of the response should only include the information from our Problem object.

> *This type of response might be appropriate for when the database is simply unreachable (there is not much the client can do about it). However, in this specific case (cannot delete the product because of related orders), a more customized response might be more appropriate.*

> *We will use a custom exception combined with IExceptionHanlderPathFeature to provide a better error response for the database integrity exception.*

7. Modify the **DeleteProduct** method in RepositoryEF to catch a **DbUpdateException** and throw a new exception with the original exception set as the inner exception.

```
public async Task<bool> DeleteProduct(Product product)
{
  try {
    Products.Remove(product);
    int rowsAffected = await SaveChangesAsync();
    return (rowsAffected > 0);
  }
  catch (DbUpdateException ex) {
    throw new ApplicationException(
      "Unable to delete the product because of an integrity constraint",
      ex);
  }
}
```


> *We should also probably log this exception. We will add that code in a future lab.*

8. Modify the **SeverError** action of ErrorController so that it returns something different when the current exception is an ApplicationException.

```
public Problem ServerError()
{
  var eh = HttpContext.Features.Get<IExceptionHandlerPathFeature>();
  var problem = new Problem();

  if (eh?.Error is ApplicationException) {
    problem.Description = eh.Error.Message;
  }
  else {
    problem.Description = "An unexpected error has occurred";
  }
  return problem;
}
```

> *If you have extra time, simulate an unavailable database. One easy way to do this is to change the database name in the connection string. Just remember to change it back after testing.*

End of Lab

Lab 11

Objectives

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

Procedures

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

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

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

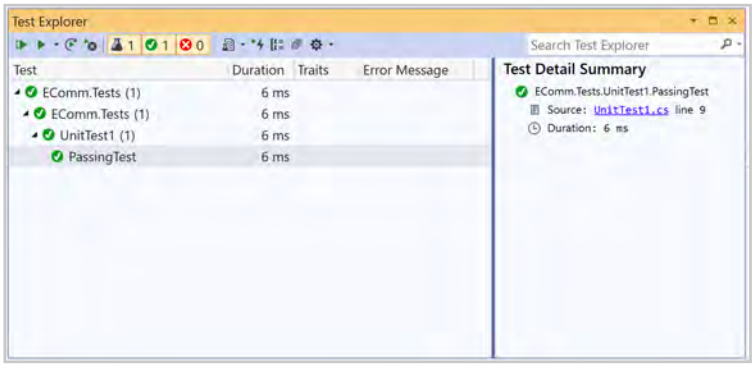
   > *We will only be adding one unit test project in this lab. However, it is very common for a solution to contain multiple unit test projects.*

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

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

4. Build the solution and then open the Visual Studio Test Runner by selecting **[Test > Test Explorer]** from the Visual Studio menu. It might take a few moments before the new test appears in the list.

5. Expand the nodes until you find **PassingTest**. Right-click on the test, click **Run**, and confirm that the test **passes**.



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

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

> *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 **Lab11** 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);

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

// Assert
Assert.IsAssignableFrom<OkObjectResult>(result);
var r = result as OkObjectResult;
Assert.IsAssignableFrom<Product>(r!.Value);
var model = r!.Value 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

Lab 12

Objectives

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

Procedures

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

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

2. Add a new folder to the EComm.API project named **Auth**.

3. Add the file **MyCustomAuthHandler.cs** from the **Lab12** folder to the Auth folder.

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

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

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

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

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

  // TODO: create ticket
}
```

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

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

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

```
var principal = new ClaimsPrincipal(
  new ClaimsIdentity(new List<Claim> {
    new Claim(ClaimTypes.Name, "SomeUser"),
    new Claim(ClaimTypes.Role, "Admin")
  }, nameof(MyCustomAuthHandler)));

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

6. **Register** the authentication handler as part of the call to **AddAuthentication** in Progam.cs.

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

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

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

8. Add the authentication middleware in the **Configure** method immediately before the authorization middleware.

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

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

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

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

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

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

11. **Run** the application and use Swagger UI to retrieve a product with a value of **abc123** provided for **X-PSK**.

12. Try is a second time with a different value for X-PSK. You should receive a **401** response.

End of Lab

Lab 13

Objectives

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

Procedures

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

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

2. Add a new project to the EComm solution named EComm.Web using the **Blazor WebAssembly App project** template.

   > *In the new project wizard, checking the box for "ASP.NET Core hosted" will create a project that uses server-side Blazor (not what we want here).*

3. Open **launchSettings.json** for the EComm.Web project and ensure the **port numbers** in the **EComm.Web** profile are set to **7100** and **5100**.

   ```
   "applicationUrl": "https://localhost:7100;http://localhost:5100",
   ```

   > *These port numbers will make sure we can run this app and the API at the same time. In production, these two apps will typically be on different machines.*

4. Set **EComm.Web** as the **startup project** and launch the application. **Explore** the three different areas by using the menu on the left side of the app.

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

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

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

6. Open **FetchData.razor** for editing.

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

7. Modify the **@code** section to reflect the EComm product API.

```
private Product[]? products;

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

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

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

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

8. Modify the **<h1>** and the **if** statement that checks for **null**.

```
<h1>Products</h1>

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

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

9. Modify the **HTML** to display **products** objects (code is on the next page).


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

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

10. Open **Program.cs** in EComm.API and add a **CORS** policy in **ConfigureServices**.

```
services.AddCors(options =>
{
  options.AddPolicy(name: "AllowedOrigins",
      builder => {
        builder.WithOrigins("http://localhost:5100",
                            "https://localhost:7100");
  });
});
```

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

11. Add code to **Configure** to enable **CORS** support. This should be somewhere **before** the call to **MapControllers**.

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

12. **Build** the solution and confirm that you do not have any compiler errors or warnings.

13. Right-click on the EComm solution, click **Set Startup Projects...**, and configure **EComm.API** and **EComm.Web** to both **Start without debugging**.

14. Select [**Debug > Start Without Debugging**] from the Visual Studio menu. Both applications should launch and will probably appear as separate tabs in the browser.

15. Click the **Fetch data** menu item in the EComm.WebApp and confirm that products are displayed.

End of Lab

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