ASP.NET Core 2 Development

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

ASP.NET Core 2.1 Development

Agenda

- Introduction
- .NET Core SDK
- ASP.NET Core Application Architecture
- Application Configuration
- Request Routing
- Models
- Controllers
- Views
- HTML Forms
- Application State

ASP.NET Core 2.1 Development

Agenda

- Data Validation
- Authentication
- Error Handling
- Logging
- Testing
- Web APIs
- Using Docker
- Deployment

ASP.NET Core 2.1 Development

Introduction

- What is .NET Core?
- .NET Core vs .NET Full Framework
- Overview of ASP.NET Core

Introduction

What is .NET Core?

- .NET Core is an open-source, cross-platform general-purpose development platform
  - Windows 7 SP1 and later
  - macOS 10.12 and later
  - RHEL, CentOS, Oracle Linux, Fedora, Debian, Ubuntu, Linux Mint, openSUSE, SLES, Alpine Linux
- github.com/dotnet/core

Introduction

What is .NET Core?

- Languages
  - C#, Visual Basic, and F#
- Frameworks
  - ASP.NET Core, UWP, Tizen

Introduction

What is .NET Core?

- .NET Core Runtime
  - Version for each platform
  - Provides assembly loading, garbage collector, and basic services
  - Includes the dotnet tool for launching applications
- ASP.NET Core Runtime
  - Includes ASP.NET Core and .NET Core runtime and framework libraries
- .NET Core SDK
  - Additional command-line tools including the language compilers
  - Includes the ASP.NET Core Runtime

Introduction

.NET Core vs .NET Full Framework

- .NET Core does not support all app-models
  - Web Forms, WCF, WPF, and Windows Forms
  - .NET Core 3 will add support for WPF and Windows Forms (on Windows platforms)
- .NET Core contains a large subset of the .NET Framework Base Class Library
  - .NET API Browser can be used to identify the availability of specific APIs
  - docs.microsoft.com/en-us/dotnet/api/

Introduction

.NET Core vs .NET Full Framework

- A higher-level framework like ASP.NET Core can run on top of .NET Core or .NET Full Framework
  - Targeting .NET Core will allow your application to be cross-platform
  - Targeting .NET Full Framework will allow you to use all of the .NET APIs and use other libraries that have a dependency on APIs that are not available in .NET Core

Introduction

Overview of ASP.NET Core

- Single web stack for Web UI and Web APIs
- Modular architecture distributed as NuGet packages
- Flexible, environment-based configuration
- Built-in dependency injection support

Introduction

Overview of ASP.NET Core

- New Features in ASP.NET Core 2.0
  - Razor Pages
  - ASP.NET Core metapackage
  - Runtime store
  - Packages target .NET Standard 2.0
  - IConfiguration instance and logging available via dependency injection by default
  - SPA templates
  - Kestrel improvements
  - And more…

Introduction

Overview of ASP.NET Core

- New Features in ASP.NET Core 2.1
  - New implementation of SignalR
  - Razor class libraries
  - Identity UI library and scaffolding
  - Improved HTTPS support
  - GDPR support
  - Integration tests
  - IHttpClientFactory
  - Kestrel default transport changed to managed sockets
  - And more…

ASP.NET Core 2.1 Development

.NET Core SDK

- Installation
- Version Management
- Command-Line Interface (CLI)
- Hello World Application

.NET Core SDK

Installation

- The .NET Core SDK is distributed using each supported platform's native install mechanism
- Available for download from www.microsoft.com/net/download
- Requires administrative privileges to install
- Information about installed SDK versions is available by using the CLI

```
dotnet --info
```


.NET Core SDK

Version Management

- By default, CLI commands use the newest installed version of the SDK
  - This behavior can be overridden through the use of a global.json file

```
{
  "sdk": {
    "version": "2.1.401"
  }
}
```

  - Will be in effect for that directory and all sub-directories beneath it

.NET Core SDK

Version Management

- Use of global.json files can allow developers to experiment with newer versions of the SDK while ensuring consistency for specific projects (e.g. include a global.json file in the source control repository)

.NET Core SDK

Version Management

- While the SDK version (tooling) is specified using a global.json file, the runtime version of .NET Core for a project is specified by the project itself

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
</PropertyGroup>
```

- It is possible, for example, to use version 2.1 of the SDK to build an application that targets the .NET Core 2.0 runtime

.NET Core SDK

Command-Line Interface (CLI)

- The .NET Core command-line interface (CLI) is a cross-platform toolchain for developing .NET applications
- Many higher-level tools and IDEs use the CLI under-the-covers
- CLI commands consist of the driver (“dotnet”), followed by a “verb” and then possibly some arguments and options

.NET Core SDK

Command-Line Interface (CLI)

- dotnet new
  - Create a new project from an available template
- dotnet restore
  - Restore the dependencies for a project (e.g. download missing NuGet packages)
- dotnet build
  - Build a project and all of its dependencies
- dotnet run
  - Run an application from its source code (performs a build if necessary)

.NET Core SDK

Command-Line Interface (CLI)

- dotnet test
  - Execute unit tests for a project
- dotnet publish
  - Pack an application and its dependencies into a folder for deployment
- And many more…

Lab 1

.NET Core SDK

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

ASP.NET Core 2.1 Development

ASP.NET Core Application Architecture

- NuGet Packages
- Application Startup
- WebHosts and Kestrel
- Middleware and the Request Pipeline
- Services and Dependency Injection

ASP.NET Core Application Architecture

NuGet Packages

- NuGet is a package manager for .NET
- All of the .NET Core and ASP.NET Core libraries (and many 3rd-party libraries) are distributed as NuGet packages
- NuGet package dependencies are stored in a project's main project file (.csproj for C#)

```
<PackageReference Include="Microsoft.EntityFrameworkCore" Version="2.1.2" />
```

ASP.NET Core Application Architecture

NuGet Packages

- Metapackages are a NuGet convention for describing a set of packages that are meaningful together
- Each framework reference (e.g. netcoreapp2.1) implicitly references a metapackage with basic functionality

ASP.NET Core Application Architecture

NuGet Packages

- ASP.NET Core 2.1 templates include a reference to the Microsoft.AspNetCore.App metapackage
  - Includes all supported ASP.NET Core and Entity Framework packages except those that contain 3rd-party dependencies
- Assets referenced by this metapackage are not deployed with the application
  - The ASP.NET Core shared framework on the target machine will contain those assets
  - By default, the application will "roll forward" to the newest installed major version of the shared framework (e.g. 2.1.0 -> 2.1.3)

ASP.NET Core Application Architecture

Application Startup

- When an ASP.NET Core application is launched, the first code executed is the application's Main method
  - The Main method configures and launches a host
- The host configures a server and request processing pipeline
  - An ASP.NET Web Host is typically constructed using an instance of IWebHostBuilder
  - CreateDefaultBuilder method constructs an IWebHostBuilder with the most common configuration and settings

```
WebHost.CreateDefaultBuilder(args).UseStartup<Startup>();
```


ASP.NET Core Application Architecture

Application Startup

- The default IWebHostBuilder:
  - Configures Kestrel as the web server
  - Sets the content root path
  - Loads configuration information from environment variables, command-line arguments, appsettings files, and user secrets
  - Configures logging
  - Enables IIS integration (when running behind IIS)
- The documentation contains information on how to customize all of the above

ASP.NET Core Application Architecture

Application Startup

- The UseStartup method of IWebHostBuilder is used to specify a type that will be used by the web host during startup

```
WebHost.CreateDefaultBuilder(args).UseStartup<Startup>();
```

- Must contain a method named Configure
  - Used to configure the app's request processing pipeline
- Can optionally include a method named ConfigureServices
  - Used to configure the app's services

ASP.NET Core Application Architecture

Hosting Environment

- ASP.NET Core reads the environment variable ASPNETCORE_ENVIRONMENT and stores the value in IHostingEnvironment.EnvironmentName
- Can be set to any string but convenience methods exist for:
  - Development
  - Staging
  - Production (default value if none specified)

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
}
```

ASP.NET Core Application Architecture

Hosting Environment

- If launchSettings.json is present, values can override environment variables

```
"IIS Express": {
  "commandName": "IISExpress",
  "launchBrowser": true,
  "environmentVariables": {
    "ASPNETCORE_My_Environment": "1",
    "ASPNETCORE_DETAILEDERRORS": "1",
    "ASPNETCORE_ENVIRONMENT": "Staging"
}
```


ASP.NET Core Application Architecture

Middleware

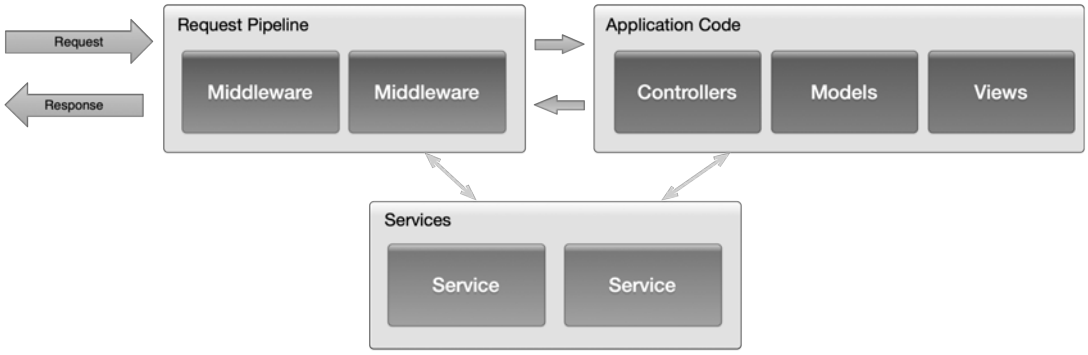
- ASP.NET uses a modular request processing pipeline
- The pipeline is composed of middleware components
- Each middleware component is responsible for invoking the next component in the pipeline or short-circuiting the chain
- Examples of middleware include…
  - Request routing
  - Handling of static files
  - User authentication
  - Response caching
  - Error handling

ASP.NET Core Application Architecture

Services

- ASP.NET Core also includes the concept of services
- Services are components that are available throughout an application via dependency injection
- An example of a service would be a component that accesses a database or sends an email message

ASP.NET Core Application Architecture



Lab 2

ASP.NET Core Application Architecture

- Create a new ASP.NET Core web application using Visual Studio 2017
- Examine the architecture of the application

ASP.NET Core 2.1 Development

Application Configuration

- Configure Method
- ConfigureServices Method
- MVC Components
- Configuration Providers and Sources
- Configuration API
- Options Pattern

Application Configuration

Configure Method

- The primary responsibility of the Configure method (in the Startup class) is to extend the request processing pipeline by adding middleware components
- The Configure method must accept a parameter of type IApplicationBuilder
  - Used to add middleware to the pipeline
- Will typically also accept an IHostingEnvironment parameter
  - Provides information such as if the application is running in development or production

Application Configuration

Configure Method

- A middleware component typically adds an extension method to IApplicationBuilder for adding it to the pipeline
  - By convention, these methods start with the prefix "Use"

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  if (env.IsDevelopment()) app.UseDeveloperExceptionPage();
  app.UseStaticFiles();

  app.Run(async (context) =>
  {
    await context.Response.WriteAsync("Hello World!");
  });
}
```

Application Configuration

ConfigureServices Method

- The optional ConfigureServices method takes a parameter of type IServiceCollection
- Services are components that are available throughout an application via dependency injection
- The lifetime of a service can be…
  - Singleton (one instance per application)
  - Scoped (one instance per web request)
  - Transient (new instance each time component requested)
- An example of a service would be a component that accesses a database or sends an email message

Application Configuration

ConfigureServices Method

- Services are typically added via extension methods available on IServiceCollection

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<ApplicationDbContext>(...);
  services.AddScoped<IEmailSender, MyEmailSender>();
  services.AddScoped<ISmsSender, MySmsSender>();
}
```

- Most methods include the service lifetime as part of the method name (e.g. AddScoped)
- The AddDbContext method is a custom method specifically for adding an Entity Framework DbContext type as a service

Application Configuration

ConfigureServices Method

- Services added in ConfigureServices are available within the application via dependency injection
- A common scenario is to follow the Explicit Dependencies Principle with controllers so that the system can automatically provide an instance of the configured service type when creating an instance of the controller

```
public class ProductController
{
  public ProductController(IEmailSender emailSender) {
    ...
  }
}
```


Application Configuration

ConfigureServices Method

- Note that ConfigureServices is called before Configure
- Allows for services to be used within the Configure method

```
public void Configure(IApplicationBuilder app, IEmailSender es)
{
  app.Run(async (context) =>
  {
    await context.Response.WriteAsync("Hello World!");
  });
  es.SendMessage("App configured");
}
```

Application Configuration

MVC Components

- For MVC-based applications, MVC services are added as part of the ConfigureServices method

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
}
```

- Adds support for authorization, formatters, views, razor, caching, data annotations, and more

Application Configuration

MVC Components

- MVC middleware is also added to the request execution pipeline as part of the Configure method
  - Allows for attribute-based routing

```
app.UseMvc();
```

- A delegate can also be passed to the UseMvc method for adding routing rules

```
app.UseMvc(routes =>
{
  routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
});
```

Application Configuration

Configuration Providers and Sources

- Before ASP.NET Core, application settings were typically stored in the application's web.config file
- ASP.NET Core introduces a completely new configuration infrastructure
  - Based on key-value pairs gathered by a collection of configuration providers that read from a variety of different configuration sources

Application Configuration

Configuration Providers and Sources

- Available configuration sources include:
  - Files (INI, JSON, and XML)
  - System environment variables
  - Command-line arguments
  - In-memory .NET objects
  - Azure Key Vault
  - Custom sources

Application Configuration

Configuration Providers and Sources

- The default IWebHostBuilder adds providers to read settings from:
  - appsettings.json
  - appsettings.{Environment}.json
  - User secrets
  - System environment variables
  - Command-line arguments

Application Configuration

Configuration Providers and Sources

- The collection and priority of configuration providers can be customized when constructing the IWebHostBuilder

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .ConfigureAppConfiguration((hostingContext, config) =>
    {
      config.AddJsonFile("mysettings.json",
                         optional: false, reloadOnChange: false);
      config.AddXmlFile("othersettings.xml",
                        optional: false, reloadOnChange: false);
      config.AddCommandLine(args);
    })
    .UseStartup<Startup>();
```

Application Configuration

Configuration API

- The configuration API provides the ability to read from the constructed collection of name-value pairs
- An object of type IConfiguration is available to be used via dependency injection

```
public class HomeController
{
  public HomeController(IConfiguration configuration)
  {
    _emailServer = configuration["EmailServer"];
  }
}
```


Application Configuration

Configuration API

- Hierarchical data is read as a single key with components separated by a colon

```
{
  "Email": {
    "Server": "gmail.com",
    "Username": "admin"
  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <section name="Email">
    <key name="Server">gmail.com</key>
    <key name="Username">admin</key>
  </section>
</configuration>
```

```
public class HomeController
{
  public HomeController(IConfiguration configuration)
  {
    _emailServer = configuration["Email:Server"];
  }
}
```

Application Configuration

Options Pattern

- The options pattern can be used to provide configuration information to other components within your application as strongly-typed objects via dependency injection

```
public class EmailOptions
{
  public string Server { get; set; }
  public string Username { get; set; }
}
```

```
public void ConfigureServices(IServiceCollection services)
{
  services.Configure<EmailOptions>(Configuration.GetSection("Email"));
}
```

```
public HomeController(IOptions<EmailOptions> emailOptions)
{
  _emailOptions = emailOptions;
}
```


Lab 3

ASP.NET Core MVC Application

- Create a new ASP.NET application that uses MVC
- Experiment with returning different content from a controller

ASP.NET Core 2.1 Development

Request Routing

- RESTful API
- Routing Middleware
- Route Templates
- Route Constraints
- MVC Middleware
- Attribute-Based Routing

Request Routing

RESTful API

- When configuring request routing, you should try to maintain a RESTful API
- Clean, extension-less URLs that identify resources
- Avoid query string parameters except for ancillary data that is related to the presentation of the information
  - Sorting key, current page number, etc.

Request Routing

Routing Middleware

- Routing middleware is responsible for mapping requests to route handlers
  - Route handlers implement IRouter
  - The default MvcRouteHandler determines the controller to instantiate and the action to invoke
- The routing system can also be used to generate URLs in view files (e.g. links)
  - Provides for easier maintenance when your routing configuration changes

Request Routing

Route Templates

- The most common way to define a route is with a route template string
- Tokens within curly braces define route value parameters which will be bound if the route is matched
  - You can define more than one route value parameter in a route segment but they must be separated by a literal value

```
site/{controller}/{action}/{id}
```

```
{language}-{country}/library/{controller}/{action}
```

```
{controller}{action}/{id}
```

Request Routing

Route Templates

- Route value parameters can have default values
  - The default value is used if no value is present in the URL for the parameter

```
{controller=Home}/{action=Index}
```

- Route value parameters may also be marked as optional
  - When bound to an action parameter, the value will be null (reference type) or zero (value type)

```
{controller=Home}/{action=Index}/{id?}
```


Request Routing

Route Templates

- The catch-all parameter (identified using an asterisk) allows for a route to match a URL with an arbitrary number of parameters

```
query/{category}/{*path}
```

```
http://localhost/query/people/hr/managers
```

```
public IActionResult Query(string category, string path)
{
  // category = "people"
  // path = "hr/managers"
}
```

Request Routing

Route Constraints

- A route value parameter can include an inline constraint
- URLs that do not match the constraint are not considered a match
- Multiple constraints can be specified for one parameter

```
products/{id:int}
```

```
products/{id:range(100, 999)}
```

```
employees/{ssn:regex(d{3}-d{2}-d{4})}
```

```
products/{id:int:range(100, 999)}
```


Request Routing

Route Constraints (Partial List)

Constraint	Example Route	Example Match
int	{id:int}	123
bool	{active:bool}	true
datetime	{dob:datetime}	2016-01-01
guid	{id:guid}	7342570B-44E7-471C-A267-947DD2A35BF9
minlength(value)	{username:minlength(5)}	steve
length(min, max)	{filename:length(4, 16)}	Somefile.txt
min(value)	{age:min(18)}	19
max(value)	{age:max(120)}	91
range(min, max)	{age:range(18, 120)}	91
alpha	{name:alpha}	Steve
regex(expression)	{ssn:regex(d{3}-d{2}-d{4})}	123-45-6789

Request Routing

Route Constraints

- Route constraints should be used to help determine the route that should be used but should not be used for the validation of input values
- If a matching route is not found, the response from the server will be an HTTP 404 (resource not found)
- Invalid input should typically result in a different response (e.g. HTTP 400 with an appropriate error message)

Request Routing

MVC Middleware

- When adding the MVC middleware, you can use one of three different methods in the Configure method...
  - UseMvc( )
    - Only supports attribute-based routing
  - UseMvc(Action<IRouteBuilder> configureRoutes)
    - Allows you to specify a callback to configure routes
  - UseMvcWithDefaultRoute( )
    - Adds a single default route

```
{controller=Home}/{action=Index}/{id?}
```

Request Routing

Attribute-Based Routing

- MVC includes attribute-based routing
- Attribute-based routing allows you to decorate a controller action with an attribute that specifies the route for that action
- Recommended when you need finer-grained control over your app's URLs

```
public class CustomerController
{
  [Route("customers/{id:int}")]
  public IActionResult Index(int id) { ... }
}
```


Request Routing

Attribute-Based Routing

- Attribute-based routing can be used in combination with centralized routing
- Attribute-based routes are added to the route table first and will take priority over centralized routes

- Hands-on lab with routing coming a bit later…

ASP.NET Core 2.1 Development

Models

- Introduction
- Persistence Ignorance
- Object-Relational Mapping
- Entity Framework (EF) Core 2

Models

Introduction

- In an MVC application, the model objects represent the data the user is interacting with
- Model data is typically retrieved by a controller and forwarded to a view for presentation (or serialized into JSON for a Web API)
- Model objects should be highly reusable and testable
- Often, it is helpful to define model objects in a separate assembly
  - Makes reuse easier and helps to maintain a clear separation of concerns

Lab 4

Models

- Create a database with some sample data
- Create a .NET Standard class library
- Add some model objects

Models

Persistence Ignorance

- The data for model objects typically comes from an external source (database, web service, file, etc.)
- For better maintainability and testability, it is a best practice for the models and controllers to not know details about where the model data comes from
- In ASP.NET Core, the data access component should be made available to controllers as a service via dependency injection
  - Can make it possible to test a controller with hard-coded data (no database)

Models

Object-Relational Mapping

- If a data access component communicates with a relational database, a necessary task will be to convert between relational data and C# objects
- This can be done manually or several frameworks exist that can help with this task
  - Entity Framework Core
  - Dapper (Micro-ORM)
  - AutoMapper (mapping one object to another)

Models

Entity Framework Core

- Entity Framework Core is a completely new version of Entity Framework for .NET Core
- Features include…
  - Modeling based on POCO entities
  - Data annotations
  - Relationships
  - Change tracking
  - LINQ support
  - Built-in support for SQL Server and Sqlite (3rd-party support for Postgres, MySQL, and Oracle)

Models

Entity Framework Core

- By creating a subclass of DbContext, EF Core can populate your model objects and persist changes

```
public class ECommContext : DbContext
{
  public DbSet<Product> Products { get; set; }
}
```

- The DbContext can be used to create a new database based on the definition of your model objects or it can work with a database that already exists (as we will do)
- The Migrations feature of EF Core can be used to incrementally apply schema changes to a database (beyond the scope of this course)

Models

Entity Framework Core

- EF Core will make certain assumptions about your database schema based on your model objects
- For example, EF Core will assume the database table names will match the name of each DbSet property

```
public class ECommContext : DbContext
{
  public DbSet<Product> Products { get; set; }
}
```

Models

Entity Framework Core

- To specify different mappings, you can use data annotations on your model objects or use EF's fluent API

```
[Table("Product")]
public class Product
{
  [Column("Name")]
  public string ProductName { get; set; }
}
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<Product>().ToTable("Product");

  modelBuilder.Entity<Product>().Property(p => p.ProductName)
                                .HasColumnName("Name");
}
```


Models

Entity Framework Core

- Objects retrieved from the context are automatically tracked for changes
- Those changes can be persisted with a call to SaveChanges

```
Product product = _context.Products(p => p.Id == id);
product.ProductName = "Something else";
_context.SaveChanges();
```

Models

Entity Framework Core

- EF Core will not automatically load related entities
- The Include method can be used to perform "eager loading" of one or more related entities

```
_dataContext.Products.Include(p => p.Supplier)
  .SingleOrDefault(p => p.Id == id);
```


Models

Entity Framework Core

- EF Core is a large topic and in-depth coverage is beyond the scope of this course
- Note that as of EF Core 2.1, there are still some missing features (when compared with the full-framework version of EF)

`docs.microsoft.com/en-us/ef/core/what-is-new/roadmap`

- Full-framework EF can still be used with an ASP.NET Core application if your application targets the full .NET Framework

Lab 5

Data Access

- Create a .NET Standard class library
- Define a subclass of DbContext
- Register the new component as a service

ASP.NET Core 2.1 Development

Controllers

- Introduction
- Requirements and Conventions
- Dependencies
- Action Results

Controllers

Introduction

- Controllers are responsible for responding to user requests
- May need to retrieve or make modifications to model objects
- Determines the appropriate response to return
  - HTML, JSON, XML, redirection, error, etc.

Controllers

Introduction

- A controller defines a set of actions that handle incoming requests
- Any public method of a controller can be an action (if a valid route to that action exists)

Controllers

Requirements and Conventions

- For a class to act as a controller, it must…
  - Be defined as public
  - Have a name that ends with Controller or inherit from a class with a name that ends with Controller
- Common conventions (not requirements) are...
  - Place all controllers in a root-level folder named Controllers
  - Inherit from Microsoft.AspNetCore.Mvc.Controller
    - Provides many helpful properties and methods

Controllers

Dependencies

- It is a recommended best practice for controllers to follow the Explicit Dependencies Principle
- Specify required dependencies via constructor parameters that can be supplied via dependency injection

```
public class HomeController
{
  private IEmailSender _emailSender;

  public HomeController(IEmailSender es) {
    _emailSender = es;
  }
}
```

Controllers

Action Results

- Although not required, controller actions typically return an instance that implements IActionResult
  - Creation of result occurs asynchronously
  - Task<IActionResult> can be used as the return type if the action itself should be asynchronous

```
public IActionResult Index()
{
  var result = new ContentResult()
  result.content = "Hello, World!";
  return result;
}
```


Controllers

Action Results

- The base class Controller provides helper methods to generate various types of results
  - View `return View(customer);`
  - Serialized object `return Json(customer);`
  - HTTP status code `return BadRequest();`
  - Raw content `return Content("Hello");`
  - Contents of a file `return File(bytes);`
  - Several forms of redirection
    - Redirect, RedirectToRoute, RedirectToAction, …
  - And more...

Controllers

Action Results

- When the return type does not implement IActionResult, a content result will be created implicitly through the use of ToString

```
public int Sum(int x, int y)
{
  return x + y;
}
```

```
public IActionResult Sum(int x, int y)
{
  int retVal = x + y;
  return Content(retVal.ToString());
}
```


Lab 6

Controllers

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

ASP.NET Core 2.1 Development

Views (Part I)

- Introduction
- Conventions
- Razor Syntax
- Layouts
- ViewData and ViewBag
- Strongly-Typed Views
- Partial Views

Views

Introduction

- In MVC, the View is responsible for providing the user interface to the client
- Transforms model data into a format for presentation to the user
- ASP.NET Core uses Razor syntax to create views that contain markup and code

Views

Conventions

- It is a common convention to place all views in a folder at the root of the project named Views
- By default, the system will look for a view at /Views/[controller]/[action].cshtml
- It is possible to specify a different view name or a full path

```
return View();
```

```
return View("AnotherView");
```

```
return View("~/Views/Stuff/SomeOtherView.cshtml");
```


Views

Conventions

- If the system cannot find a view in the default location, it will also look in a folder under Views named Shared
- The Shared folder is a convenient place to put views that are used by more than one controller
  - Layouts
  - Error pages
  - Reusable partial views
- If unable to locate a view, an exception will be thrown

Views

Conventions

- Views that exist only to be used by other views are typically given a name that begins with an underscore
  - ViewStart files `_ViewStart.cshtml`
  - Layouts `_Layout.cshtml`
  - Partial views `_ProductList.cshtml`

Views

Razor Syntax

- Very often, the content of a view needs to be generated dynamically
- Razor syntax allows you to embed C# code within a view
- Recommended best practice is to limit the code in a view to code specific to data presentation
  - Too much logic within a view creates something that is difficult to read, maintain, and test
- Controllers should deliver data to a view in a form that is ready for presentation

Views

Razor Syntax

- The key transition character in Razor is @
  - Used to transition from markup to code
- Razor parser uses a look-ahead algorithm to determine the end of a code expression
- Visual Studio editor displays text Razor interprets as code with a darker background color

```
<ul>
@foreach (Course course in Model) {
  <li>@course.Number is named @course.Title.</li>
}
</ul>
```


Views

Razor Syntax

- The output from a Razor expression is automatically HTML encoded
- Disable using Html.Raw( )

```
<span>@Html.Raw(course.Description)</span>
```

Views

Razor Syntax

- Sometimes, Razor needs a little help to identify the end of a code expression
- Use @(to force Razor to interpret all text as code until it encounters the closing parentheses

```
<span>@course.Price * 0.10</span>
```

```
<span>@(course.Price * 0.10)</span>
```


Views

Razor Syntax

- To render @ into the response when Razor thinks it's code, escape the character with a second one

```
<span>Follow @acme on Twitter</span>
```

```
<span>Follow @@acme on Twitter</span>
```

Views

Razor Syntax

- A stand-alone code block can be specified using { }
- Statements within a code block must end with a semi-colon

```
@{
  int i = 5;
  i++;
}
```

- Variables declared in a code block are scoped to the page

Views

Razor Syntax

- A razor comment is identified using @* and *@

```
@*
  This is a comment
*@
```

- Razor comments are server-side comments
  - They are not sent to the client

Views

Layouts

- Layouts help maintain a consistent look and feel across multiple views
- Defines a common template for some or all of your views
- Call to RenderBody( ) in a layout marks the location where the content of the individual view will be rendered

```
<div class="container body-content">
  @RenderBody()
  <hr />
  <footer>
    <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
  </footer>
</div>
```


Views

Layouts

- The layout for a view can be specified in the view itself

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

- Can also be specified in a _ViewStart file
  - Code within a _ViewStart file is executed before the code in any view within the same folder

Views

Layouts

- A layout can also define sections that are required or optional
- Layout defines where a section appears
- Page specifies the content for the section

```
<body>
  @RenderBody()
  @RenderSection("Footer")
</body>
```

```
<p>This is the body content</p>

@section Footer {
  <span>The footer!</span>
}
```


Views

View Data

- An object of type ViewDataDictionary is used to pass data from a controller to a view
- Controllers and views each have properties that allow easy access to the data in the ViewDataDictionary
  - ViewData
  - ViewBag
  - Model

Views

View Data

- ViewData is a key-value store of type <string, object>

```
public ActionResult Index()
{
  ViewData["Title"] = "About";
}
```

```
<title>@ViewData["Title"]</title>
```

- Can be useful for small pieces of information (e.g. page title)

Views

View Data

- More complex objects can be stored in ViewData
- Will typically require type casting in the view and is therefore discouraged

```
public ActionResult Detail(int id)
{
  ViewData["Product"] = GetProduct(id);
}
```

```
<h3>@((ViewData["Product"] as Product).ProductName)</h3>
```

Views

ViewBag

- ViewBag is a dynamic variable that provides access to ViewData
- Does not provide any additional features other than a cleaner syntax
- ViewData and ViewBag can be used interchangeably

```
public ActionResult Index()
{
  ViewData["Message"] = "Hello";
}
```

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


Views

Strongly-Typed Views

- A major disadvantage of ViewData is the the lack of design-time support when creating views (Intellisense) and the need for type-casting within the view
- A better approach would be to specify a type for the view's Model property
- This creates a strongly-typed view

Views

Strongly-Typed Views

- To create a strongly-typed view, use a model directive as the first line of code in the view

```
@model IEnumerable<EComm.Data.Product>
```

- To avoid having to use fully-qualified type name, you can add a using directives to the _ViewImports file

```
@using EComm.Data
```


Views

Strongly-Typed Views

- Use an overload of the View convenience method to pass a model object to a strongly-typed view
- Use the Model property of the view to access the object in the view

```
return View(products);
```

```
@foreach (var product in Model) {
  <p>@product.ProductName</p>
}
```

Views

Partial Views

- A partial view is a reusable piece of view content that can be shared between different views
- Provides an effective way of breaking up a large view into smaller components
- A partial view is defined in the same manner as a normal view and can be strongly-typed
- Rendered into another view by using the Partial tag helper
  - Model object passed to the partial view with the for attribute

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


Lab 7

Views (Part I)

- Modify the home page to display a list of products

ASP.NET Core 2.1 Development

Views (Part II)

- View Models
- HTML and URL Helpers
- Tag Helpers
- View Components
- Client-Side Dependencies
- Razor Pages

Views

View Models

- Sometimes, a model and a view do not match up exactly
  - View may require more data than is present in the model
  - View may only be displaying a portion of the model object
- Additional data could be sent using ViewData
- Another option is to create an additional object that is specifically designed to be the model for the view
  - This object is commonly referred to as a view model

Views

View Models

- Common uses of a view model include…
  - Multiple model objects of different types
  - Model object plus property value choices (select lists)
  - Addition of web-specific artifacts that you do not want to add to your model class

Views

View Models

- A view model can be implemented a few different ways…
  - Inherit from a model class
  - Contain a model object
  - Reimplement the properties of the model object that it represents
- Each approach has advantages and drawbacks

Views

View Models

- Some members of the community believe that you should never send a model directly to a view
- Every view should be strongly-typed to a view model type
- This can provide some benefits but can also result in much more work in some cases

Views

View Models

- One powerful use of view models is to create a hierarchy of view model classes that match your view hierarchy
- Define your layout to be strongly-typed to your view model base class
  - View model base class can contain things that are common to all views (title, description, etc.)
- Define the views based on your layout to be strongly-typed to a subclass of the layout's view model

Views

Helpers

- ASP.NET Core provides a collection of helpers that you can use when when authoring views
  - HTML Helpers
  - URL Helpers
  - Tag Helpers
- Helpers make it easier to generate common pieces of view content and help with maintenance by dynamically generating content based on things like the routing configuration or model fields

Views

HTML Helpers

- HTML Helpers were introduced with ASP.NET MVC
- Provided as a collection of extension methods
- Used to dynamically generate HTML elements

```
@Html.ActionLink("Create New", "Create", "Course")
```

```
<a href="/Course/Create">Create New</a>
```

Views

HTML Helpers

Helper	Description
ActionLink	Renders a hyperlink element
BeginForm	Marks the start of a form
CheckBox	Renders a check box
DropDownList	Renders a drop-down list
Hidden	Renders a hidden form field
ListBox	Renders a list box
Password	Renders a text box for entering a password
RadioButton	Renders a radio button
TextArea	Renders a text area (multi-line text box)
TextBox	Renders a text box

Views

HTML Helpers

- HTML Helpers that generate content for model data will sometimes accept a lambda expression to specify the model property
- All strongly-typed helpers end with For
- Some helpers can even dynamically choose the HTML element to use based on the type of the model property

```
<div>
  @Html.LabelFor(model => model.FirstName)
</div>
<div>
  @Html.EditorFor(model => model.FirstName)
  @Html.ValidationMessageFor(model => model.FirstName)
</div>
```

Views

URL Helpers

- URL Helpers can be useful for generating outbound links based on the current routing configuration

```
<a class="btn" href="@Url.Action("Index", "Department")">Learn more</a>
```


Views

Custom Helpers

- To create a custom helper, define a new extension method for HtmlHelper or UrlHelper
- Extension methods are static methods in a static class that use the this keyword to tell the compiler the type being extended
- If generating HTML, return an HtmlString to prevent automatic encoding by the view engine

```
public static HtmlString Image(this IHtmlHelper html,
                               string src, string alt)
{
  string str = String.Format("<img src=\"{0}\" alt=\"{1}\" />",
                             src, alt);
  return new HtmlString(str);
}
```

Views

Tag Helpers

- ASP.NET Core introduces a new feature called Tag Helpers
- Tag Helpers provide the ability to generate markup in a cleaner, more HTML-friendly way compared to HTML Helpers
- Server-side concerns are specified using attributes that begin with asp-

```
@Html.LabelFor(model => model.FirstName, new {@class="caption"})
```

```
<label class="caption" asp-for="FirstName"></label>
```


Views

Tag Helpers

Helper	Description
a	Renders a hyperlink element
cache	Caching the enclosed content (defaults to 20 minutes)
distributed-cache	Uses an implementation of IDistributedCache provided via dependency injection
environment	Renders content based on the specified hosting environment
form	Renders an HTML form tag
img	Renders an image tag with optional automatic versioning
input	Renders an HTML input element for a model property
label	Renders an HTML label element for a model property
partial	Renders a partial view
select	Renders an HTML select element for a model property and list of choices
textarea	Renders an HTML textarea element

Views

Tag Helpers

- Tag Helpers that generate a URL allow for additional route data to be specified using attributes that begin with asp-route-

```
<a asp-controller="Product" asp-action="Detail"
   asp-route-id="@product.Id">@product.ProductName</a>
```

```
<a href="/product/detail/5">Bananna</a>
```

```
<a asp-controller="Greeting" asp-action="SayHello"
   asp-route-myname="Bill">Greet Me!</>
```

```
<a href="/greeting/sayhello?myname=Bill">Greet Me!</a>
```


Lab 8

Views

- Add the ability to display the details for a product

Views

View Components

- Sometimes, it can be helpful for part of a view's content to come from the execution of code
- Helps to maintain separation of responsibilities and enhance reusability
- A view component is a separate class that typically inherits from ViewComponent and implements an InvokeAsync method

```
public async Task<IViewComponentResult> InvokeAsync()
{
  var products = _context.Products.ToList();
  return View(products);
}
```


Views

View Components

- View components support dependency injection
- You can use a view component within a view by calling Component.InvokeAsync

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

- You can also use it like a tag helper as long as you add the assembly in the _ViewImports file

```
<vc:product-list>
</vc:product-list>
```

```
@addTagHelper *, EComm.Web
```

Views

View Components

- It is possible to pass arguments to a view component's InvokeAsync method

```
@Component.InvokeAsync("ProductList", new { sort = "Price", page = 1 })
```

```
<vc:product-list sort="Price" page="1">
</vc:product-list>
```


Views

View Components

- View components can exist in any project folder or namespace
- When returning a view, the recommended path for views is Views/Shared/Components/[view component name]/[view name]
- The default view name for a view component is Default (not Index)

Lab 9

Views

- Refactor the product list to be a view component

Views

Client-Side Dependencies

- When creating a modern web application, you will typically have a number of client-side frameworks that your views use
  - jQuery
  - Bootstrap
  - Angular
  - React
  - Many more…

Views

Client-Side Dependencies

- Client-side frameworks typically consist of many different resources
  - JavaScript
  - CSS
  - Images
  - Fonts
- You can include these files as part of your application or use another source to provide them (e.g. CDN)

Views

Client-Side Dependencies

- Many tools exist for managing 3rd-party client-side dependencies
  - Bower, npm, and Yarn are some examples
- A new option provided by Microsoft is the Microsoft Library Manager (LibMan)
  - github.com/aspnet/LibraryManager
  - To enable in a Visual Studio project, right-click on the project and select [Manage Client-Side Libraries…]
  - To add a new client-side library, right-click on the project and select [Add > Client-Side Library…]

Views

Client-Side Dependencies

- LibMan uses a file named libman.json
- Supports intellisense in Visual Studio
- Right-click on libman.json to perform specific operations

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.3.1",
      "destination": "wwwroot/lib/jquery",
      "files": [
        "jquery.min.js"
      ]
    }
  ]
}
```


Views

Client-Side Dependencies

- ASP.NET Core views also have the ability to conditionality include client-side files based on environment name
- A version string can also be automatically appended to file names to avoid caching issues

```
<environment names="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment names="Staging,Production">
  <link rel="stylesheet"
     href="https://ajax.aspnetcdn.com/.../bootstrap.min.css" />
  <link rel="stylesheet" href="~/css/site.min.css"
     asp-append-version="true" />
</environment>
```

Views

Client-Side Dependencies

- A fallback feature can be used to handle the case where a remote source is unavailable

```
<environment names="Staging,Production">
  <link rel="stylesheet"
    href="https://ajax.aspnetcdn.com/.../bootstrap.min.css"
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
    asp-fallback-test-class="sr-only"
    asp-fallback-test-property="position"
    asp-fallback-test-value="absolute" />
</environment>
```


Views

Razor Pages

- Razor Pages is a feature introduced as part of ASP.NET Core 2
- Makes coding page-focused scenarios easier and more productive
- The MVC services includes support for Razor Pages

```
public void ConfigureServices(IServiceCollection services)
{
  // Includes support for Razor Pages and
  services.AddMvc();
}
```

Views

Razor Pages

- Including the @page directive in a view file makes the file into an MVC action
  - A separate controller is no longer needed
- A @model directive can be used to specify a class designed to provide data to the view

```
@page
@model MyPageModel
<h2>Separate page model</h2>
<p>@Model.Message</p>
```


Views

Razor Pages

```
public class MyPageModel : PageModel
{
  public string Message { get; private set; } = "PageModel in C#";

  public void OnGet()
  {
    Message += $" Server time is {DateTime.Now}";
  }
}
```

Lab 10

Routing and Errors

- Use attribute-based routing
- Use an inline constraint
- Experiment with a variety of invalid requests

ASP.NET Core 2.1 Development

HTML Forms

- Introduction
- Form Tag Helper
- Form Submissions
- Model Binding

HTML Forms

Introduction

- When working with HTML forms in a web application, there are two high-level operations to deal with…
  - Generate the form for presentation to the user
  - Handle the submitted data (including validation)

HTML Forms

Form Tag Helper

- The Form Tag Helper in ASP.NET Core…
  - Generates the HTML action attribute for an MVC controller action or named route
  - Generates a hidden request verification token to prevent cross-site request forgery (CSRF)

```
<form asp-controller="Department" asp-action="Edit" method="post">
  ...
</form>
```

```
<form method="post" action="/department/edit">
  <input name="__RequestVerificationToken" type="hidden"
         value="..." />
  ...
</form>
```

Lab 11

HTML Forms

- Create the product edit form

HTML Forms

Form Submissions

- When configuring the routing for an action, an action selector can be used to specify an HTTP verb
- Allows for more than one controller action with the same name (and route) if different verbs are used
- In the case of a form, an HTTP GET is used to retrieve the form while an HTTP POST is typically used to receive the submission

```
public IActionResult Edit(int id) { ... }

[HttpPost]
public IActionResult Edit(Product product) { ... }
```

HTML Forms

Form Submissions

- To check the request verification token, the ValidateAntiForgeryToken attribute must also be applied to the POST action

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(Product product) { ... }
```


HTML Forms

Model Binding

- When a form is submitted, the model binding system attempts to populate the parameters of the action with values in the request
  - Form fields
  - Route value
  - Query strings
- Items above are listed in priority order (i.e. form field values will take precedence over other values)

HTML Forms

Model Binding

- If an action accepts an object parameter, the model binding system will create an instance of that type and attempt to populate its public properties with values from the request
- If validation errors occur during the model binding process, the IsValid property of the ModelState property will return false

```
public ActionResult Edit(ProductViewModel vm)
{
  if (ModelState.IsValid) { ... }
  ...
}
```


HTML Forms

Model Binding

- It is important to ensure the model binding system does not alter values that you do not intend to be modified
  - Can lead to a security vulnerability known as over-posting
- Attributes can be used to define properties that should not participate in model binding

```
[BindNever]
public int EmployeeId { get; set; }
```

- Alternatively, use a view model that only includes properties that are intended to participate in model binding

Lab 12

HTML Forms

- Complete the ability to edit a product

ASP.NET Core 2.1 Development

Data Validation

- Introduction
- Data Annotations
- Model Binding
- Input Tag Helpers
- Validation Tag Helpers

Data Validation

Introduction

- Whenever any data from the client is being used to perform an action, it is important to have data validation in place
  - Don't skip validation for hidden form fields, HTTP header values, cookies, etc. (all are easy to modify)
- Client-side validation provides a good user experience and improved application scalability (less trips to the server)
- Server-side validation must also be provided
  - Client-side validation is easy to circumvent or may not be supported on the client

Data Validation

Data Annotations

- A variety of data annotations can be added to the model (or view model) that is sent to a view
- Data annotations are looked for by helpers (to enable client-side validation) and during model binding (to perform server-side validation)

```
public class ProductEditViewModel
{
  [Required]
  public string ProductName { get; set; }
```

Data Validation

Data Annotations

Attribute	Purpose
[Required]	Property value is required (cannot allow nulls)
[StringLength]	Specifies a maximum length for a string property
[Range]	Property value must fall within the given range
[RegularExpression]	Property value must match the specified expression
[Compare]	Property value must match the value of another property
[EmailAddress]	Property value must match the format of an email address
[Phone]	Property value must match the format of a phone number
[Url]	Property value must match the format of a URL
[CreditCard]	Property value must match the format of a credit card number

Data Validation

Input Tag Helpers

- The Input Tag Helper (and other helpers) generate HTML based on a model expression

```
<input asp-for="ProductName" />
```

- Will set the HTML type attribute based on the type of the model expression and any data annotations
- Generates HTML5 validation attributes for any relevant data annotations

```
<input data-val="true"
      data-val-required="The ProductName field is required"
      id="ProductName" name="ProductName" value="Syrup" />
```

Data Validation

Model Binding

- Data annotations are also used during the model binding process
- If a value is considered to be invalid, an error is added to ModelState and ModelState.IsValid will return false
- ModelState is also looked at by helpers in the view

Data Validation

Validation Tag Helpers

- The Validation Message Tag Helper displays a message for a single property on your model

```
<span asp-validation-for="Email" />
```

- The Validation Summary Tag Helper displays a summary of validation errors
  - Can display individual property errors as well as model-level errors

```
<div asp-validation-summary="ValidationSummary.ModelOnly"></div>
```

Data Validation

IValidatableObject

- For custom server-side validation, you can implement the IValidatableObject interface for the type being populated by the model binder
- Any errors returned are automatically added to ModelState by the model binder

```
public IEnumerable<ValidationResult> Validate(ValidationContext
                                                validationContext)
{
  var retVal = new List<ValidationResult>();
  if (BirthDate > HireDate) {
    retVal.Add(new ValidationResult("Employee cannot be
                                     hired before they were born"));
  }
  return retVal;
}
```


Lab 13

Data Validation

- Add data validation for editing a product

ASP.NET Core 2.1 Development

Application State

- Introduction
- HttpContext.Items
- Session State
- TempData

Application State

Introduction

- There are several options for handling state in ASP.NET Core
- Options that roundtrip data to the client include…
  - Query string values
  - Hidden form fields
  - Cookies
- Options that involve data stored on the server include…
  - HttpContext.Items (scoped to the request)
  - Session
  - Cache (shared for all users)

Application State

Introduction

- Per-user server-side state should be avoided when possible to maintain the scalability of your application
- However, caution should be used when sending state to the client
  - Can be read and possibly modified
  - Increases the bandwidth used for each request

Application State

HttpContext.Items

- HttpContext provides a property of type Dictionary<object, object> named Items
- Available during the processing of a request and then discarded
- Middleware could add something to Items that is available later within a controller

```
app.Use(async (context, next) =>  {
  // perform some verification
  context.Items["isVerified"] = true;
  await next.Invoke();
});
```

```
var v = HttpContext["isVerified"];
```

Application State

Session State

- ASP.NET Core includes a package that provides middleware for managing session state
  - Microsoft.AspNetCore.Session
- Session uses an IDistributedCache implementation
  - ASP.NET Core includes implementations for in-memory, Redis, and SQL Server
- By default, session uses a cookie named .AspNet.Session to send the session Id to the client

Application State

Session State

- Session must be configured in your Startup class

```
services.AddDistributedMemoryCache();
services.AddSession();
```

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

Application State

Session State

- Session state is made available via a property of HttpContext named Session that implements ISession
- Session always accepts and stores byte[ ]

```
HttpContext.Session.Set("username", Encoding.UTF8.GetBytes(username));
```

```
byte[] data;
bool b = HttpContext.Session.TryGetValue("username", out data);
if (b) username = Encoding.UTF8.GetString(data);
```

- More complex objects must be serialized into a byte[ ]
  - One option is to first convert the object into a JSON string

Application State

TempData

- Another state management option is something called TempData
- The goal of TempData is to provide per-user state that lives for one additional request
- Helpful when implementing the POST-Redirect-GET pattern
- ASP.NET Core 1.0 used session to provide this
- ASP.NET Core 1.1 added a cookie-based TempData provider

Lab 14

Application State

- Implement shopping cart functionality (Part 1)

Lab 15

Application State

- Implement shopping cart functionality (Part 2)

Lab 16

Application State

- Implement shopping cart functionality (Part 3)

ASP.NET Core 2.1 Development

Authentication

- Introduction
- ASP.NET Core Identity
- Cookie Middleware
- Authorization
- Claims-Based Authorization

Authentication

Introduction

- There is a wide variety of authentication options available for an ASP.NET Core web application
- Covering all of the available authentication options and variations is beyond the scope of this course
- We will focus primarily on a forms-based authentication system that uses claims-based authorization

Authentication

ASP.NET Core Identity

- ASP.NET Core Identity is a full-featured membership system available for ASP.NET Core
- Supports username/password login as well as external login providers such as Facebook, Google, Microsoft Account, Twitter and more
- Can use SQL Server or a custom credential store
- The Visual Studio templates that include authentication use ASP.NET Core Identity

Authentication

Cookie Middleware

- ASP.NET Core provides cookie middleware
- Serializes a user principal into an encrypted cookie
- Validates incoming cookie, recreates the principal and assigns it to the User property of HttpContext
- ASP.NET Core Identity uses the cookie middleware but you can also use it as a standalone feature

Authentication

Cookie Middleware

- The cookie middleware is available via the Microsoft.AspNetCore.Authentication.Cookies NuGet package
- First step is to configure the authentication middleware service

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddAuthentication(
      CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie();
```

Authentication

Cookie Middleware

- To enable the authentication middleware, it must be added to the request processing pipeline

```
public void Configure(IApplicationBuilder app)
{
  ...
  app.UseAuthentication();
```


Authentication

Cookie Middleware

- Once authenticated, you can construct a ClaimsPrincipal which can be used to construct the authentication cookie
- The ClaimsPrincipal contains a ClaimsIdentity that contains collection of claims
- A claim is a simple string pair (type, value)
  - Some claim types are pre-defined but any string can be used

```
var principal = new ClaimsPrincipal(
  new ClaimsIdentity(new List<Claim>
  {
    new Claim(ClaimTypes.Name, lvm.Username),
    new Claim(ClaimTypes.Role, "Admin"),
    new Claim("MyType", "MyValue")
  },
  CookieAuthenticationDefaults.AuthenticationScheme));
```

Authentication

Cookie Middleware

- The SignInAsync method is used to construct an authentication cookie from a principal

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

- The SignOutAsync method is also available

```
await HttpContext.SignOutAsync(
  CookieAuthenticationDefaults.AuthenticationScheme);
```


Authentication

Authorization

- The Authorize attribute can be used to authorize access to specific functionality
- Can be applied at the action or controller level
- If nothing else is specified, the attribute simply ensures the current user has been authenticated
- The AllowAnonymous attribute can be used to opt-out an action

```
[Authorize]
public class AdminController : Controller
{
  public IActionResult Dashboard() { ... }

  [AllowAnonymous]
  public IActionResult Login() { ... }
}
```

Authentication

Claims-Based Authorization

- With claims-based authorization, you can define a policy that specifies what claims must be present for a user to be authorized

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

- The Authorize attribute can then be used to enforce the policy

```
[Authorize(Policy="AdminsOnly")]
public IActionResult Dashboard() { ... }
```


Authentication

Claims-Based Authorization

- Programmatic authorization checks can also be performed within an action to enforce function-level access control

```
public IActionResult Dashboard()
{
  foreach (var claim in User.Claims)
  {
    // decide what the user should see
  }
```

Lab 17

Authentication

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

ASP.NET Core 2.1 Development

Error Handling

- Best Practices
- HTTP Error Status Codes
- Status Code Pages
- Developer Exception Page
- Exception Filters

Error Handling

Best Practices

- Handle errors as best you can when they occur
- Record the error information and/or send a notification
- Provide the user with an appropriate response
  - Do not reveal information that a malicious user could potentially use against you (e.g. database schema information)
  - Give the user some options (e.g. link to visit the home page in the case of a 404)
  - Use static content whenever possible to avoid an error page that itself produces an error

Error Handling

HTTP Error Status Codes

- The HTTP protocol defines a range of status codes that signify an error
  - 4xx = client error (not found, bad request)
  - 5xx = server error
- It is a best practice to define an appropriate customized response that will be returned to the client in these cases

Error Handling

Status Code Pages

- The StatusCodePage middleware can be used to define the response that should be returned for HTTP error status codes
- By default, this middleware will return a simple string describing the error

```
public void Configure(IApplicationBuilder app,
                      IHostingEnvironment env)
{
  app.UseStatusCodePages();
}
```


Error Handling

Status Code Pages

- The StatusCodePage middleware can also…
  - Use a custom function to provide a response

```
app.UseStatusCodePages(context =>
  context.HttpContext.Response.SendAsync("Status code: " +
    context.HttpContext.Response.StatusCode, "text/plain"));
```

  - Redirect (302) the user to a different page

```
app.UseStatusCodePagesWithRedirects("~/errors/{0}");
```

  - Return the HTTP error status code but with the results from a different page

```
app.UseStatusCodePagesWithReExecute("~/errors/{0}");
```

Error Handling

Developer Exception Page

- When an exception occurs during development, it is helpful to get as much information as possible about the error
- You can add middleware that provides a developer exception page in the case of an uncaught exception
- It is important to make sure this page is only used in the development environment

```
public void Configure(IApplicationBuilder app,
                      IHostingEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
}
```


Error Handling

Exception Filters

- In an MVC-based application, filters can be used to execute code before and/or after an action executes
- Helpful for cleanly applying default exception handling behavior (logging and notifications)
- Filters can be configured at the action, controller, or global level
- An exception filter can be used to handle exceptions that occur during controller creation and model binding

```
public class DepartmentController : SocratesController
{
  [MyExceptionFilter]
  public IActionResult Index()
  { ... }
}
```

Error Handling

Exception Filters

- Define an exception filter by creating a subclass of ExceptionFilterAttribute

```
public class MyExceptionFilterAttribute : ExceptionFilterAttribute
{
  public override void OnException(ExceptionContext context)
  {
    if (!env.IsDevelopment()) { return; }
    var result = new ViewResult { ViewName = "CustomError" };

    // provide exception data to view if desired

    context.Exception = null;  // mark exception as handled
    context.Result = result;
  }
}
```


Error Handling

Exception Filters

- Global filters are added using an overload of the AddMvc method within the ConfigureServices method

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc(options => {
    options.Filters.Add(typeof(MyExceptionFilter));
  });
}
```

Lab 18

Error Handling

- Configure status code pages

ASP.NET Core 2.1 Development

Logging

- Introduction
- Configuration
- ILogger

Logging

Introduction

- Just as important as error handling is the ability to record information about events that occur
- Logging of error information is essential for tracking down an issue that occurs in production
- It is sometimes helpful to record information about events that are not errors
  - Performance metrics
  - Authentication audit logs

Logging

Introduction

- ASP.NET Core has a logging API that works with a variety of logging providers
- Built-in providers allow you to log to the console and the Visual Studio Debug window
- Other 3rd-party logging frameworks can be used to provide other logging options
  - Serilog
  - NLog
  - Log4Net
  - Logentries
  - Loggr

Logging

Configuration

- Any component that wants to use logging can request an ILogger<T> as a dependency

```
public class ProductController : Controller
{
  public ProductController(ECommDataContext dataContext,
    ILogger<ProductController> logger) { }
}
```


Logging

Configuration

- ILogger defines a set of extension methods for different verbosity levels
  - Trace (most detailed)
  - Debug
  - Information
  - Warning
  - Error
  - Critical

```
_logger.LogInformation("About to save department {0}", id);
```

ASP.NET Core 2.1 Development

Testing

- Introduction
- Unit Testing
- xUnit
- Testing Controllers
- Integration Testing

Testing

Introduction

- Testing your code for accuracy and errors is at the core of good software development
- Testability and a loosely-coupled design go hand-in-hand
- Even if not writing tests, keeping testability in mind helps to create more flexible, maintainable software
- The inherit separation of concerns in MVC applications can make them much easier to test

Testing

Introduction

- Unit testing
  - Test individual software components or methods
- Integration testing
  - Ensure that an application's components function correctly when assembled together

Testing

Unit Testing

- A unit test is an automated piece of code that involves the unit of work being tested, and then checks some assumptions about a noticeable end result of that unit

Testing

Unit Testing

- A unit of work is the sum of actions that take place between the invocation of a public method in the system and a single noticeable end result by a test of that system
- A noticeable end result can be observed without looking at the internal state of the system and only through its public API
  - Public method returns a value
  - Noticeable change to the behavior of the system without interrogating private state
  - Callout to a third-party system over which the test has no control

Testing

Unit Testing

- Good unit tests are…
  - Automated and repeatable
  - Easy to implement
  - Relevant tomorrow
  - Easy to run
  - Run quickly
  - Consistent in its results
  - Fully isolated (runs independently of other test)

Testing

Unit Testing

- A unit test is typically composed of three main actions
  - Arrange objects, creating and setting them up as necessary
  - Act on the object
  - Assert that something is as expected

Testing

Unit Testing

- Often, the object under test relies on another object over which you have no control (or doesn't work yet)
- A stub is a controllable replacement for an existing dependency in the system
  - By using a stub, you can test your code without dealing with the dependency directly
- A mock object is used to test that your object interacts with other objects correctly
  - Mock object is a fake object that decides whether the unit test has passed or failed based on how it is used

Testing

xUnit

- A test project is a class library with references to a test runner and the projects being tested
- Several different testing frameworks are available for .NET
  - Visual Studio includes the ability to add a roject that use the MSTest framework or the xUnit framework
- xUnit has steadily been gaining in popularity inside and outside of Microsoft

Testing

xUnit

- Fact attribute is used to define a test which is always true
- Theory attribute is used to define which is true for a particular set of data

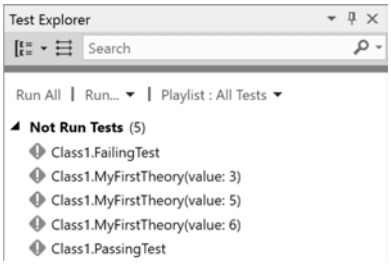
```
[Fact]
public void PassingTest()
{
  Assert.Equal(4, Add(2, 2));
}
```

```
[Theory]
[InlineData(3)]
[InlineData(5)]
[InlineData(6)]
public void MyFirstTheory(int value)
{
  Assert.True(IsOdd(value));
}
```

Testing

xUnit

- Tests can be run using the Visual Studio Test Explorer



- Tests can also be run by using the .NET Core command line interface

```
> dotnet test
```


Testing

Testing Controllers

- When looking to test a controller, ensure that all dependencies are explicit so that stubs and mocks can be used when needed
- When testing a controller action, check for things like…
  - What is the type of the response returned?
  - If a view result, what is the type of the model?
  - What does the model contain?

Testing

Integration Testing

- Integration tests check that an app functions correctly at a level that includes the app's supporting infrastructure
  - Request processing pipeline
  - Database
  - File system

Testing

Integration Testing

- ASP.NET Core's WebApplicationFactory class is used to create a TestServer and an HttpClient

```
public class BasicTests :
  IClassFixture<WebApplicationFactory<EComm.Web.Startup>>
{
    private readonly WebApplicationFactory<EComm.Web.Startup> _factory;

    public BasicTests(WebApplicationFactory<EComm.Web.Startup> factory)
    {
        _factory = factory;
    }
}
```

Testing

Integration Testing

- ASP.NET Core's WebApplicationFactory class is used to create a TestServer and an HttpClient

```
[Theory]
[InlineData("/")]
[InlineData("/Index")]
[InlineData("/About")]
public async Task Get_EndpointsSuccessAndContentType(string url)
{
  // Arrange
  var client = _factory.CreateClient();

  // Act
  var response = await client.GetAsync(url);

  // Assert
  response.EnsureSuccessStatusCode(); // Status Code 200-299
  Assert.Equal("text/html; charset=utf-8",
    response.Content.Headers.ContentType.ToString());
}
```


Lab 19

Testing

- 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

ASP.NET Core 2.1 Development

Web APIs

- Introduction
- Retrieval Operations
- Create Operations
- Update Operations
- Delete Operations
- Bad Requests
- Cross-Origin Request Sharing (CORS)

Web APIs

Introduction

- ASP.NET Core provides extensive support for building Web APIs
- The separate Microsoft "Web API" framework has been merged into ASP.NET Core
  - The Controller base class is used to handle both scenarios – there is no longer a separate ApiController class

Web APIs

Retrieval Operations

- In a Web API, retrieval operations are performed with an HTTP GET request
- If successful, the response should use an HTTP 200 status code
- When returning an ObjectResult, ASP.NET Core will automatically format the response as JSON

```
return new ObjectResult(product);
```


Web APIs

Create Operations

- In a Web API, create operations are performed with an HTTP POST request
- If successful, the response should use an HTTP 201 (created) status code with a Location header set to the URI of the newly created item
- The CreatedAtAction and CreatedAtRoute methods can be used to generate a correctly formatted response

```
return CreatedAtAction("Get", new { id = product.Id }, product);
```

Web APIs

Update Operations

- In a Web API, update operations are performed with an HTTP PUT request
- If successful, the response should use an HTTP 204 (no content) status code

```
return new NoContentResult();
```


Web APIs

Delete Operations

- In a Web API, delete operations are performed with an HTTP DELETE request
- If successful, the response should use an HTTP 204 (no content) status code

```
return new NoContentResult();
```

Web APIs

Bad Requests

- Your Web API should return an HTTP 400 (bad request) when the request is not formed correctly (e.g. missing payload for a create request)

```
return BadRequest();
```


Web APIs

Cross-Origin Resource Sharing (CORS)

- Browser security prevents a web page from making ajax requests to another domain
- CORS is a W3C standard that allows a server to relax this policy
- A server can explicitly allow some cross-origin requests
- CORS is configured in ASP.NET Core via a service and middleware

```
services.AddCors();
```

```
app.UseCors(builder =>
    builder.WithOrigins("http://example.com"));
```

Lab 20

Web API

- Build a Web API for product data

ASP.NET Core 2.1 Development

Using Docker

- Advantages of Containerized Applications
- Docker Fundamentals
- Microsoft ASP.NET Core Docker Images
- Running a Container
- Visual Studio Docker Support
- AWS and Azure

Using Docker

Advantages of Containerized Applications

- Docker is an open platform that enabled developers and administrators to build images, ship, and run applications in a loosely isolated environment called a container
- Developer – Helps me to eliminate the "works on my machine" problem
- Administrator – Allows me to treat hardware instances less like "pets" and more like "cattle"

Using Docker

Docker Fundamentals

- The Docker platform uses the Docker engine to build and package apps as Docker images
- Docker images are created using files written in the Dockerfile format

Using Docker

Microsoft ASP.NET Core Docker Images

- Microsoft provides a collection of official images to act as the starting point for your own images
  - Have the .NET Core runtime pre-installed
  - Some have the .NET Core SDK installed and can be used as a build server
- Available on Docker Hub
  - microsoft/dotnet
  - microsoft/aspnetcore

Using Docker

Running a Container

- A container is a running instance of an image
- When running an ASP.NET Core application in a container, it is necessary to map the internal container port to a port on the host machine

```
docker run –it --rm -p 8000:80 ecomm/website
```

Using Docker

Visual Studio Support

- Visual Studio 2017 supports building, running, and debugging containerized ASP.NET Core applications
  - Must have Docker for Windows installed
- You can enable Docker support when creating a project
- You can also add Docker support to an existing app by selecting [Add > Docker Support]

Using Docker

AWS and Azure

- Both Amazon AWS and Microsoft Azure have extensive support for hosting containers
  - AWS EC2 Container Service and Container Registry
  - Azure Container Service and Container Registry

ASP.NET Core 2.1 Development

Deployment

- Page and View Compilation
- Publishing
- Reverse Proxies

Deployment

Page and View Compilation

- Razor views are compiled at runtime when the view is invoked
- It is also possible to compile Razor views ahead of time and deploy them with the app (precompilation)
  - Disabled by default in ASP.NET Core 1.0 and 1.1
  - Enabled by default in ASP.NET Core 2.0 and later

Deployment

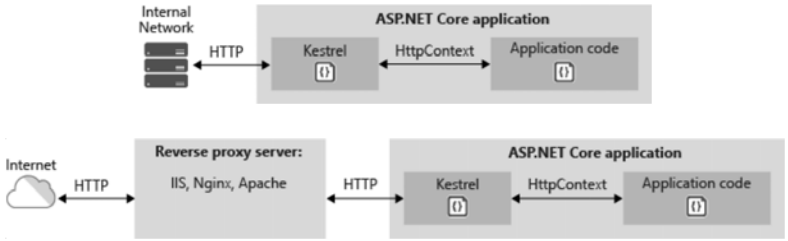
Publishing

- The dotnet publish command compiles app code and copies the files needed to run the app into a publish folder
- An app can be published as a self-contained or framework-dependent app
  - If self-contained, the publish folder will contain the .NET runtime

Deployment

Host and Web Server

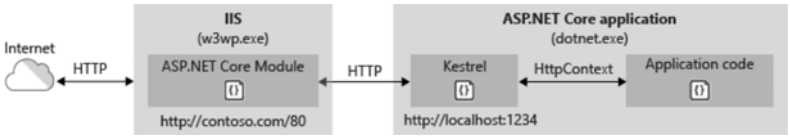
- In a production environment, it is common practice to use another web server as a reverse proxy server in front of Kestrel



Deployment

Host and Web Server

- When deploying to IIS, the AspNetCoreModule hooks into the IIS pipeline and redirects traffic to Kestrel
  - Available as part of the ASP.NET Core Server Hosting Bundle (VS installs this module into IIS Express automatically)



Introduction to ASP.NET Core Development

ASP.NET Core 2.1 and Visual Studio

About this Lab Manual

This lab manual consists of a series of hands-on lab exercises for learning to build ASP.NET Core web applications that target .NET Core 2.1

System Requirements

- **Visual Studio 2017/2019 (any edition)** with the ASP.NET Core tools installed. If using Visual Studio 2017, update 15.7 (or later) must be installed
  - Instructions for adding components to an existing installation of Visual Studio are available at <https://docs.microsoft.com/en-us/visualstudio/install/modify-visual-studio>
  - Visual Studio 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 1

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 (standard Windows command prompt 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
```

> *There 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 MyFirstProject
```

> *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 **MyFirstProject.csproj**

PowerShell: `> cat MyFirstProject.csproj`

Command Prompt: `> type MyFirstProject.csproj`

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>
</Project>
```

> *Within this file, the target framework should be specified as something like "netcoreapp2.1". 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 MyFirstProject
{
  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 **current 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 MyFirstWebApp
```

> *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. Examine the contents of **MyFirstWebApp.csproj**.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
</Project>
```


This file is very similar to the file created during the previous lab for the console application. There is one additional package dependency specified (Microsoft.AspNetCore.App) and specifies that a folder (wwwroot) should be included in the build output.

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

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

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>();
```

Even though this is a web application, it is in fact still a console application packaged as a dll with a Main method (just like in the previous lab). The Main method here is used to configure a "WebHost" that will listen for incoming HTTP requests.

The static CreateWebHostBuilder 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.

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

```
app.Run(async (context) =>
{
  await context.Response.WriteAsync("Hello World!");
});
```

We will discuss the Configure and ConfigureServices methods later on in the course. However, for now, just know that this code specifies that the string "Hello World!" should be written to the response for any incoming request.

15. **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).

16. 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.*

17. Make a request to something like **http://localhost:5000/more/in?the=path** and notice that the response from the application is the same.

> *This happens because the application right now is simply configured to return the same response for EVERY received request regardless of anything additional provided as part of the request.*

18. Make a request to **https://localhost:5001** (don't forget the "s") and examine the message displayed by the browser.

> *The issue here is that ASP.NET Core is using a self-signed certificate that web browsers will not trust by default. Later on, we will resolve this issue so that we can easily test our application using HTTPS.*

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

End of Lab

Lab 2

Objectives

- Create a new ASP.NET Core web application using **Visual Studio 2017**
- Examine the **architecture** of the application

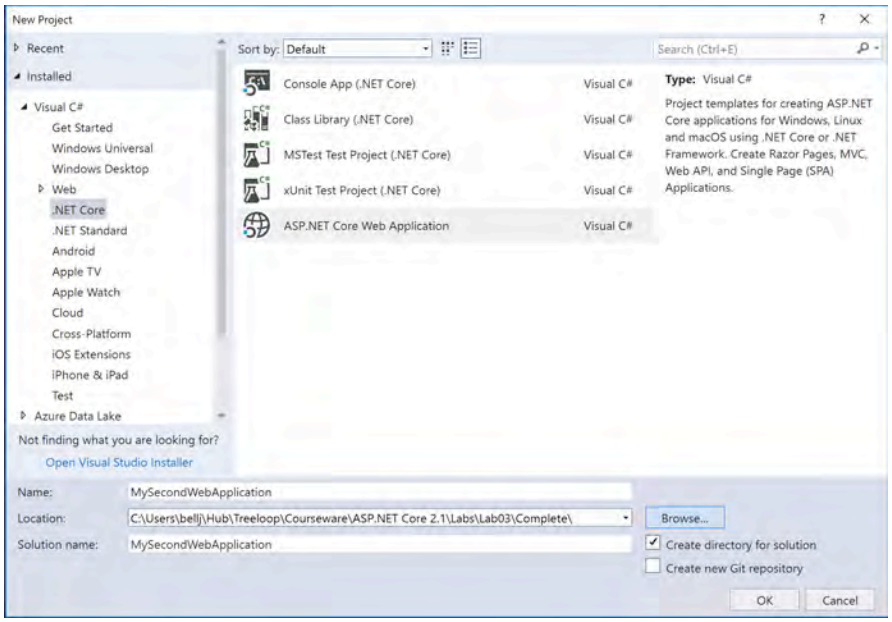
Procedures

1. Open Visual Studio 2017, choose **[File > New > Project...]**, select the **.NET Core** category under **Visual C#**, and select the template named **ASP.NET Core Web Application**

   a. Name the project **MySecondWebApplication**

   b. Choose an appropriate **location** for the project

   c. Click **OK**



2. In the next dialog, make the following selections:

   a. Select **.NET Core** and **ASP.NET Core 2.1** in the dropdown boxes at the top of the dialog.

> *The first dropdown is where you can choose to run ASP.NET Core on top of the full .NET Framework.*

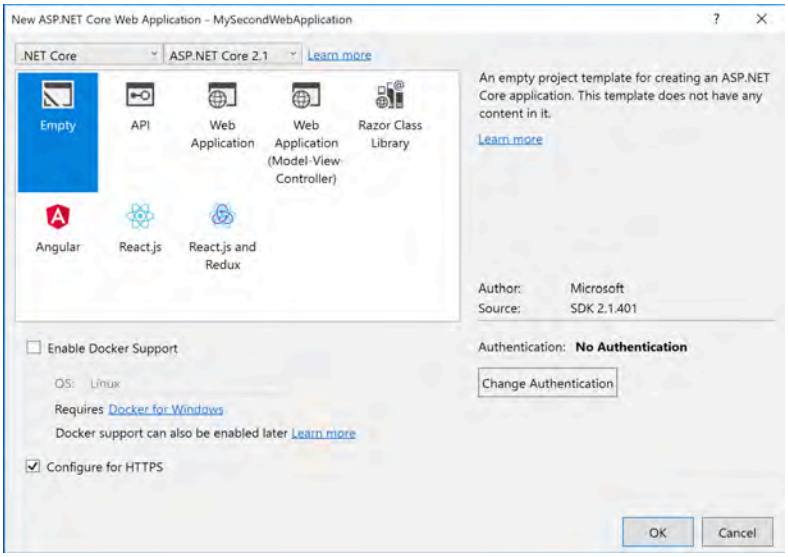
b. Select the **Empty** template.

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 **OK**

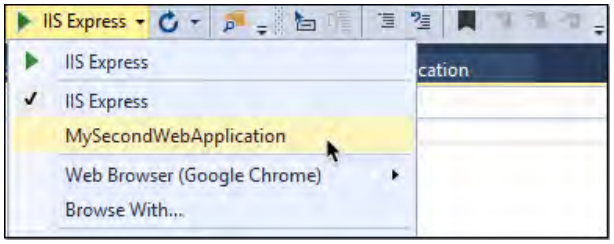


> *You will have an opportunity to experiment with Visual Studio's Docker support and with authentication later on in the course.*

4. **Examine** the files that were created. They should look very similar to what was created when you used the CLI in the previous lab.

a. A new file named **launchSettings.json** is visible under the Properties node in Solution Explorer. This file contains information used by Visual Studio.

> *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 **IIS Express** as the **debug target**. You may get a dialog asking if you would like to trust the IIS Express SSL certificate. This allows you to avoid the browser warning we received in the previous lab when using HTTPS.

6. Confirm the application functions (displays "Hello World!") and then close the browser.

7. Run the application a second time with **MySecondWebApplication** selected as the **debug target**. You may be prompted about trusting a certificate again since the application will be running outside of IIS and using a different certificate. Notice that a console window appears similar to when we ran the application from the command line in the previous lab.

End of Lab

Lab 3

Objectives

- Create a new ASP.NET application that uses **MVC**
- **Experiment** with returning **different content** from a controller

Procedures

1. Create a **new project** in Visual Studio 2017.
   a. Choose the **ASP.NET Core Web Application** template.
   b. Name the project **EComm.Web** and name the solution **EComm**
   c. Ensure **ASP.NET Core 2.1** is selected for the version and select the **Web Application (Model-View-Controller)** template.
   d. Do **not** select any authentication or Docker support.
2. **Examine** the files that are included in the project. Differences from the project you created in the previous lab include:
   a. An **appsettings.json** and **appsettings.Development.json** file has been added to the project.
   b. The **Startup** class includes code that obtains an **IConfiguration** object, sets up some middleware, and adds some services.
   c. A **Controllers** folder has been added with a class named **HomeController**.
   d. A **Views** folder has been added with a collection of **cshtml** view files.
   e. A **Models** folder has been added with a class for representing an error.
   f. The **wwwroot** folder contains a collection of client-side resources.
3. **Run** the application and click "Accept" for the cookie notification if it appears at the top of the page.
4. Use the menu at the top of the page to view the **about** and **contact** pages. Make a note of what the URL looks like for each page.
5. Stop the application and open the code for the **HomeController** class.

6. **Experiment** with returning **different results** from the Index method instead of the default "return View( );". Here are a few things to try:

```
return Content("Hello from HomeController");

return Content("<em>Hello</em> from HomeController", "text/html");

return Content("{\"Greeting\":\"Hello\"}", "application/json");
```

> *As you can see, this method can return any content you wish (including acting like a web API and returning JSON). Of course, there are other ways to return HTML and JSON that are much more pleasant to write and maintain.*

7. After you are done experimenting, change the code back to "**return View();**"

End of Lab

Lab 4

Objectives

- Create a **database** with some **sample data**
- Create a .NET Standard **class library**
- Add some **model objects**

Procedures

1. If not already open, re-open your **EComm** solution from Lab 3.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 4.*

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

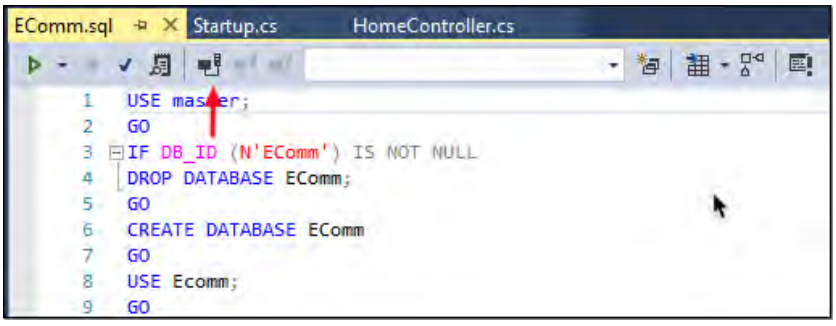
3. If you already have an entry for **(localdb)\MSSQLLocalDB**, skip to **step 4**.

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

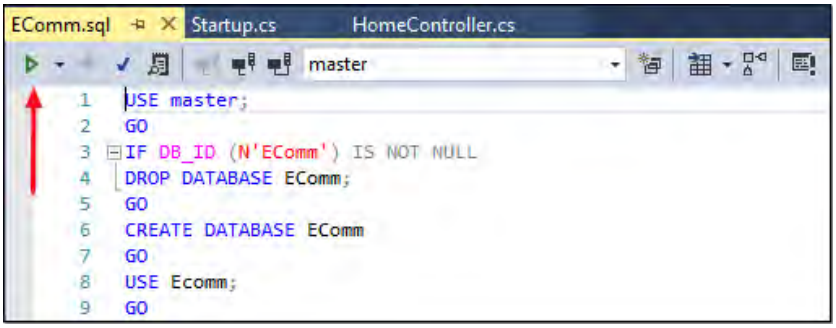
   b. Expand the **Local** node, select **MSSQLLocalDB**, and click **Connect.**

4. From the Visual Studio menu, select [**File > Open > File...**] and select **EComm.sql** from the course's **Code** folder.

5. Click the **Connect** button in the toolbar of the query window, select **MSSQLLocalDB** (under Local) and click **Connect**.



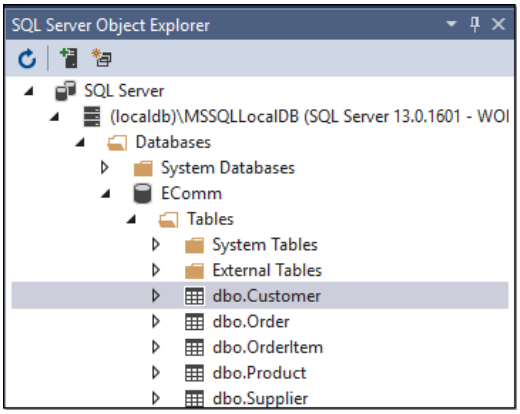
6. Click the **Execute** button in the toolbar of the query window to run the script.



> *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.*

7. Right-click on **(localdb)\MSSQLLocalDB** in SQL Server Object Explorer and click **Refresh**.

8. Right-click on the **Customer** table of the new **EComm** database and select **View Data**. You should see a collection of customer records.



> *The next step will be to create a class library project that will contain the model objects that will be populated from the database data.*

9. Right-click on the **EComm** solution and select **[Add > New Project...]**

10. Under the **.NET Standard** category, select the **Class Library (.NET Standard)** template and name the project **EComm.Model**

> *Since we want our model objects to be reusable, putting them in a .NET Standard-based library will allow this library to be used in a wide variety of .NET projects (.NET Core, Full Framework, Xamarin, and more).*

11. **Delete** the **Class1.cs** file from the new project.

> *Since the model objects don't contain anything specific to ASP.NET Core, the completed source files have been provided for you in the Labs folder.*

12. Right-click on the **EComm.Model** project, choose **[Add > Existing Item...]**, and add all of the files from the **Code\Lab04** directory in the Labs folder.

13. **Examine** the code for the files that you just added.

14. Right-click on the **Dependencies** node in the **EComm.Web** project, choose **[Add Reference...]**, and select the **EComm.Model** project.

15. **Build** the solution. You should not have any errors or warnings.

End of Lab

Lab 5

Objectives

- Create a .NET Standard **class library**
- Define a subclass of **DbContext**
- Register the new component as a **service**

Procedures

1. If not already open, re-open your **EComm** solution from Lab 4.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 5.*

2. Right-click on the **EComm** solution and select **[Add > New Project...]**

3. Under the **.NET Standard** category, select the **Class Library (.NET Standard)** template and name the project **EComm.DataAccess**

4. **Delete** the **Class1.cs** file from the new project.

   > *Since EF Core is not included in a class library project by default, we will need to add it via NuGet.*

5. Right-click on the **EComm.DataAccess** project and select **[Manage NuGet Packages...]**

6. Make sure **Browse** is selected at the top of the NuGet Package Manager window, search for **Microsoft.EntityFrameworkCore.SqlServer**, and install the latest stable version.

   > *Since the data access library will be working with our model types, we will need to add a reference to the EComm.Model project.*

7. Right-click on the **Dependencies** node in the **EComm.DataAccess** project, choose **[Add Reference...]**, and select the **EComm.Model** project.

8. Add a **new class** to the **EComm.DataAccess** project named **ECommDataContext**.

9. 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
```

From now on, we will assume that you will add appropriate using directives as they are required. They will now longer be explicitly mentioned in the lab instructions.

10. Add a **constructor** to the ECommDataContext class that accepts a parameter of type **DbContextOptions** and calls the equivalent base class constructor.

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

This is necessary because we will want to be able to specify some options when registering this type as a service (e.g. the database connection string).

11. Add a public property of type **DbSet** for each model type.

```
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; }
```

12. Add an override of **OnModelCreating** and use a loop to change the name of the table each model type is mapped to.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  foreach (var entity in modelBuilder.Model.GetEntityTypes()) {
    entity.Relational().TableName = entity.ClrType.Name;
  }
}
```

We have to do this because our database table names are singular (e.g. Customer) while our property names are plural (e.g. Customers). This loop tells EF to use the type name for the table name instead of the DbSet property name.

The next step will be to configure our web application so that it can use the data access library to work with our database data.

13. Right-click on the **Dependencies** node in the **EComm.Web** project, choose **[Add Reference...]**, and select the **EComm.Model** and **EComm.DataAccess** projects.

14. Open the **appsettings.json** file in the EComm.Web project and add a **ConnectionStrings** section with a connection string for the EComm database. The connection string itself should be on **one line**.

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

We can now register the DataContext as a service in our web application so that it can be used throughout our application via dependency injection.

15. Open the **Startup.cs** file in the EComm.Web project and register the DataContext type within the **ConfigureServices** method.

```
services.AddDbContext<ECommDataContext>(
  options => options.UseSqlServer(
  Configuration.GetConnectionString("ECommConnection")));
```

The AddDbContext method is a specialized method for registering a DbContext type as a service. It provides the ability to provide some options and defaults to a scoped service (this can be changed by passing a service lifetime to AddDbContext).

16. **Build** the solution and address any errors or warnings.

End of Lab

Lab 6

Objectives

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

Procedures

1. If not already open, re-open your solution from Lab 5.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 6.*

2. Open **HomeController.cs** in EComm.Web and add a **private field** of type **ECommDataContext**.

```
public class HomeController : Controller
{
  private ECommDataContext _dataContext;
```

3. Add a **constructor** to HomeController that accepts a **DataContext** and sets the private field.

```
public HomeController(ECommDataContext dataContext)
{
  _dataContext = dataContext;
}
```

4. Modify the **Index** action so that it returns some information that is obtained from the database.

```
public IActionResult Index()
{
  return Content($"Number of products: {_dataContext.Products.Count()}");
}
```

5. **Run** the application and check the results. If a SqlException is thrown, double-check the accuracy of the connection string in appsettings.json.

6. Modify the action to return a **list of product objects** and test the behavior.

```
public List<Product> Index()
{
  return _dataContext.Products.ToList();
}
```

> *When simply returning a list of objects, ASP.NET Core decided to serialize the list and return the data in JSON format. 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). We will take advantage of this capability later on in the course when we build a proper Web API.*

7. Change the Index action back to returning a **View**.

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

End of Lab

Lab 7

Objectives

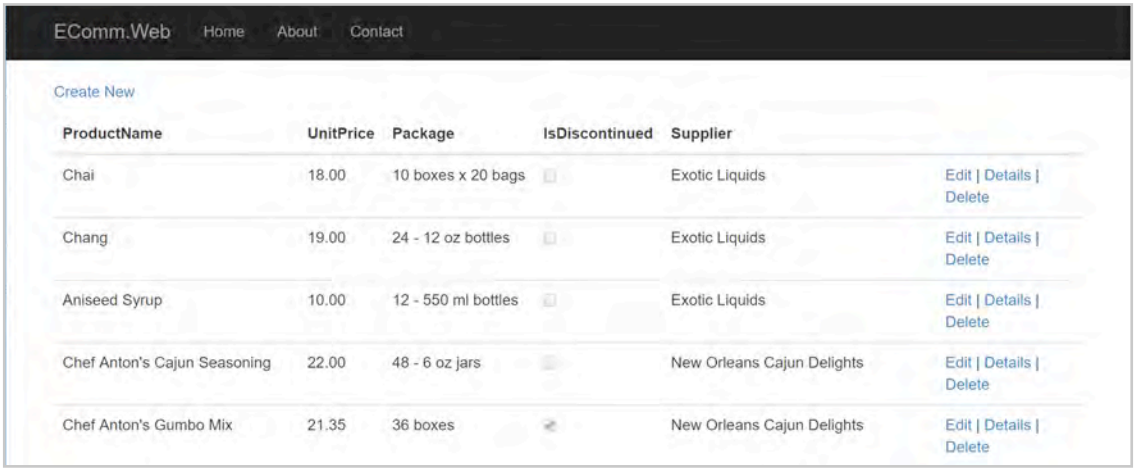
- Modify the home page to display a **list of products**

Procedures

1. If not already open, re-open your solution from Lab 6.

> *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 7.*

2. Use what you have learned so far to modify the homepage so that it appears similar to the screenshot below (see the items under the screenshot for additional requirements).



   a. The list itself should be in a **partial view** (right-click on a view folder and select [**Add > View...**] to generate a good starting point).

   b. Take note of the **Supplier** column. This is data from a related entity.

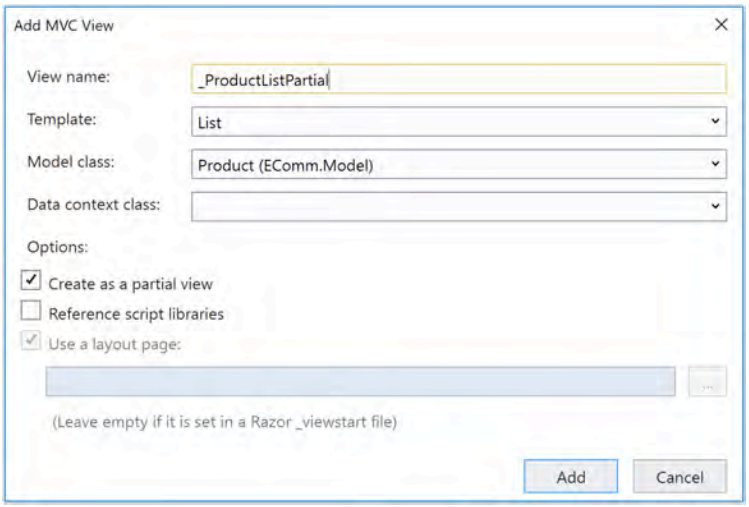
   c. Step-by-step instructions are provided on the next page but try your best to accomplish the task without using the instructions.

   d. Once your application satisfies the requirements, think about what you would do to make it better.

3. Right-click on the **Views/Shared** folder and click [**Add > View...**].

> *This view could be placed in the Views/Home folder. However, if think the view might be used in more than one parent view, placing it in the Shared folder makes that easier.*

4. Configure the view to be a **partial view** named **_ProductListPartial** that uses the **List** template for the **Product** model class (we do not need the script libraries).



5. Modify the **Views/Home/Index** view so that it is **strongly typed** and displays the partial view (all of the other markup can be deleted).

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

<br/>
<partial name="_ProductListPartial" />
```

6. Modify the **Index** action of **HomeController** so that is passes a **list of products** to the view.

```
public IActionResult Index()
{
  var products = _dataContext.Products.Include(p => p.Supplier).ToList();
  return View(products);
}
```

> *Notice the use of the Include method to ensure that we have the related supplier object for each product.*

7. Run the application and check the results. Some things aren't quite right (when compared to the requirements)
   a. The **Id** field of each product is being shown. We don't want that.
   b. The **SupplierId** is being shown instead of the name of the supplier.
8. Open **Views/Shared/_ProductListPartial.cshtml** for editing.
9. **Remove** the **header** and **column** for the product's **Id**.
10. Modify the **header** of the **supplier** column.

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

11. Modify the **supplier** column to use the **CompanyName** of the supplier.

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

12. **Run** the application and check the results.

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 solution from Lab 7.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 8.*

2. Use what you have learned so far to modify the product list so that each product name appears as a link that goes to a page that displays the details for that product.

   a. The links in the product list should be generated by using the **anchor tag helper**.

   b. The product detail page should be provided by a new **ProductController**.

   c. Like the list, you should not display the product's Id and you should display the company name of the supplier.

   d. Once again, step-by-step instructions are provided on the next page but you should try your best to accomplish the task without using the instructions.

3. Add a new controller named ProductController and configure it like HomeController so that it can receive an ECommDataContext via DI.

```
public class ProductController : Controller
{
  private ECommDataContext _dataContext;

  public ProductController(ECommDataContext dataContext)
  {
    _dataContext = dataContext;
  }
}
```

4. Add a new subfolder under **Views** named **Product** and add a view file to that folder. Name the view **Detail** that is **strongly-typed** to **Product** and use the **Details** template (make sure you do not create the view as a partial view).

5. Change the **Supplier** property similar to what you did for the product list.

```
<dt>
  @Html.DisplayNameFor(model => model.Supplier)
</dt>
<dd>
  @Html.DisplayFor(model => model.Supplier.CompanyName)
</dd>
```

6. Add a **Detail** action to **ProductController**. The action should take an **id** parameter of type int and retrieve the proper product with its related supplier.

```
public IActionResult Detail(int id)
{
  var product = _dataContext.Products.Include(p => p.Supplier)
                .SingleOrDefault(p => p.Id == id);
  return View(product);
}
```

> *Notice that we are not checking here whether the product is null (i.e. the id provided does not match an existing product). We will address this in a future lab.*

7. Modify **_ProductListPartial.cshtml** so that an anchor tag helper is used to display the product names as links to the product's detail page.

```
<tr>
  <td>
    <a asp-controller="Product" asp-action="Detail"
       asp-route-id="@item.Id">@item.ProductName</a>
  </td>
```

8. **Run** the application and test the functionality of the links.

End of Lab

Lab 9

Objectives

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

Procedures

1. If not already open, re-open your solution from Lab 8.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 9.*

2. Use what you have learned so far to change the product list from being a partial view (used by the home page) into a self-contained **view component** that can be used anywhere in the application.

   a. Put the new view component a folder named **ViewComponents** at the same level as the Controllers folder.

   b. Remember the **view search path** used by view components.

   c. Use **tag helper syntax** to render the view component.

   d. Once again, step-by-step instructions are provided on the next page but you should try your best to accomplish the task without using the instructions.

3. Add a **new folder** to the EComm.Web project named **ViewComponents**.

4. Add a new class to the ViewComponents folder named **ProductList** that inherits from **ViewComponent** and uses same technique as ProductController to receive an ECommDataContext via DI.

```
public class ProductList : ViewComponent
{
  private ECommDataContext _dataContext;

  public ProductList(ECommDataContext dataContext)
  {
    _dataContext = dataContext;
  }
}
```

5. Add an **InvokeAsync** method that uses code similar to what is currently in HomeController's Index action.

```
public Task<IViewComponentResult> InvokeAsync()
{
  var products = _dataContext.Products.Include(p => p.Supplier).ToList();
  return Task.FromResult<IViewComponentResult>(View(products));
}
```

6. Create a folder in the proper location for the view component (**Views/Shared/Components/ProductList**).

7. Move **_ProductListPartial.cshtml** into the new folder and rename it to **Default.cshtml**.

8. Add a directive to **_ViewImports.cshtml** so that we can use the view component using tag helper syntax.

```
@addTagHelper *, EComm.Web
```

9. Replace the partial tag helper in **Index.cshtml** with a tag helper that uses the view component and remove the @model directive at the top.

```
<br/>
<vc:product-list />
```

10. Modify the **Index** action of **HomeController** so that it simply returns a view.

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

11. **Run** the application and check the results.

End of Lab

Lab 10

Objectives

- Use **attribute-based routing**
- Use an **inline constraint**
- Experiment with a variety of **invalid requests**

Procedures

1. If not already open, re-open your solution from Lab 9.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 10.*

   > *The first objective of this lab will be to change the format of the URL used to request the details for a product. Right now, an example request would look like [/product/detail/4]. Instead, we would like the same request to be in the form [/product/4]*

2. Add an **attribute-based route** to the **Detail** action of **ProductController**.

```
[Route("product/{id}")]
public IActionResult Detail(int id)
```

3. **Run** the application and click on one of the product links on the home page.

   > *Notice that everything still works as it should but the URL appears differently. This is because we did not hard-code any URLs into the HTML for the product list. The Tag Helpers we are using generate the URLs dynamically based on the routing configuration.*

4. Return to the **Detail** action and add an **inline constraint** to the route that specifies that the id route parameter must be an **integer**.

```
[Route("product/{id:int}")]
public IActionResult Detail(int id)
```

5. **Run** the application and confirm that everything still works. Then try the following **experiments** and take note of the **response** you receive in each case:

   a. Make a request with a **controller name** that **does not exist** (e.g. /person/index)

   b. Make a request with a controller name that exists but with an **action name** that **does not exist** (e.g. /product/edit)

   c. Make a request that would be valid but add **extra URL components** (e.g. /home/index/5/stuff)

   d. Make a request that does not match the **route constraint** (e.g. /product/abc)

   e. Change a controller action to return a **view** that **does not exist** and make a request for that action. For example, temporarily modify the Index action in HomeController so it appears like what is shown below (make sure to change it back after the test).

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

   f. Make a request for a **product** that **does not exist** in the database (e.g. /product/9999)

In the first four examples above, the issue is with the client. The client is making a request for something that does not exist (no valid route). Therefore, in each case, the response from the application is an HTTP 404 (resource not found).

In the fifth case (e), the issue is with the application. The client made a perfectly valid request. However, during the execution of the controller action, a component required for generating the response (a view file) was missing. Therefore, an exception was thrown and an HTTP 500 (internal server error) was returned. This seems correct.

In the final case, the page displays an empty product. (or may thrown an exception depending on the code in the view). This is not right. Let's fix that.

6. Modify the **Detail** action of **ProductController** so that an **HTTP 404** is returned when the product requested does not exist.

```
[Route("product/{id:int}")]
public IActionResult Detail(int id)
{
  var product = _context.Products.Include(p => p.Supplier)
                .SingleOrDefault(p => p.Id == id);

  if (product == null) return NotFound();

  return View(product);
}
```

7. **Run** the application and repeat the experiment performed in step **5f**. The response should now be a 404.

> *Although a 404 is the correct status code to return in this case, we are leaving it up to the client (browser) to decide on what user experience to provide. Later on in the course, we will take control over the user experience for errors.*

End of Lab

Lab 11

Objectives

- Create the **product edit form**

Procedures

1. If not already open, re-open your solution from Lab 10.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 11.*

2. Use what you have learned so far to create the form for editing a product.

   a. The **supplier** field should be represented as a **drop-down list** of supplier company names.

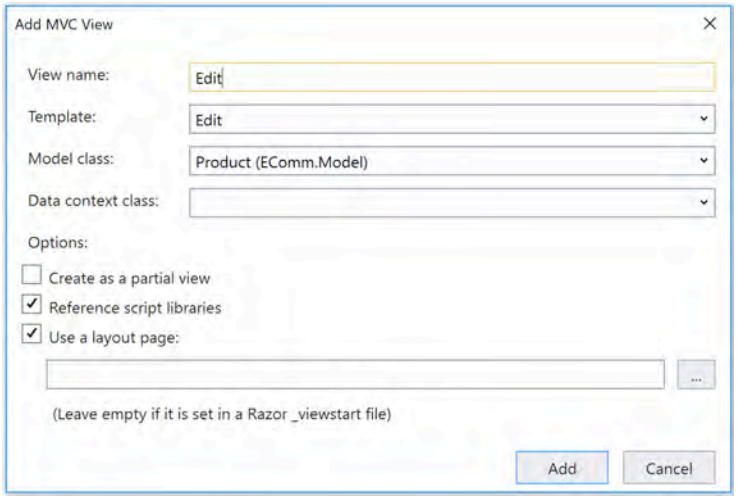
   b. Use a **view model** to provide the list of suppliers to the view. Place the view model into a folder named **ViewModels** at the same level as the Controllers folder.

   c. Make sure to check the box to **reference script libraries** when adding the edit view.

   d. You do **not** need to handle the submitted data at this point (that will be the next lab).

   e. Once again, step-by-step instructions are provided on the next page but you should try your best to accomplish the task without using the instructions.

3. Add a new view to the **/Views/Product** folder named **Edit**. Select the Edit template and be sure to check the box to **reference script libraries**.



4. Add a method to ProductController named **Edit** that accepts an **id** parameter of type int. Retrieve the appropriate product and send it to the view.

```
public IActionResult Edit(int id)
{
  var product = _dataContext.Products
                .SingleOrDefault(p => p.Id == id);

  if (product == null) return NotFound();

  return View(product);
}
```

5. Modify the **edit links** in the product list (now in the the view for the view component) and on the detail page so that they pass the correct value for the product's **id**.

```
<a asp-controller="Product" asp-action="Edit"
   asp-route-id="@item.Id">Edit</a>
```

6. **Run** the application and test the form.

> *The form should load with the product's data. However, we should not allow the user to edit the Id property and the supplier field needs to be addressed. If you were to click the Save button, the form will simply reload since we have yet to create the POST action.*

7. Remove the element (and all of its content) containing the field for the **Id** property and add the Id value to the **form tag helper.**

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

> *Instead of including the Id value as part of the form URL, we could of kept the Id form field but made it hidden. We just need to make sure the Id is somewhere in the request.*

8. Create a new folder named **ViewModels** at the same level as the Controllers folder.

9. Add a new class to the ViewModels folder named **ProductEditViewModel**.

10. Define ProductViewModel as follows:

```
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 List<SelectListItem> Suppliers { get; set; }
}
```

11. Change the Edit view to be **strongly-typed** to a ProductEditViewModel.

```
@model EComm.Web.ViewModels.ProductEditViewModel
```

12. Modify the element for SupplierId to use the **select tag helper** for a list of the **supplier company names**.

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

13. Modify the **Edit** action of ProductController to construct and pass a ProductEditViewModel to the view (instead of the product).

```
var suppliers = _dataContext.Suppliers.ToList();
var pvm = new ProductEditViewModel
{
  Id = product.Id, ProductName = product.ProductName,
  UnitPrice = product.UnitPrice, Package = product.Package,
  IsDiscontinued = product.IsDiscontinued,
  SupplierId = product.SupplierId, Suppliers = suppliers.Select(
    s => new SelectListItem
    { Text = s.CompanyName, Value = s.Id.ToString() }).ToList()
};
return View(pvm);        }
```

14. **Run** the application and check the appearance of the form.

End of Lab

Lab 12

Objectives

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

Procedures

1. If not already open, re-open your solution from Lab 11.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 12.*

2. Use what you have learned so far to handle the data submitted by the product edit form and save the changes to the database.

   a. You can use the same view model type to accept the incoming data or create a separate one.

   b. Check the **ModelState.IsValid** property before saving the product.

   c. After a successful save, redirect the user to the home page.

   d. Once again, step-by-step instructions are provided on the next page but you should try your best to accomplish the task without using the instructions.

3. Add a new action to ProductController named **Edit** for HTTP **POST** that accepts an incoming ProductEditViewModel.

```
[HttpPost]
public IActionResult Edit(ProductEditViewModel pvm)
{ }
```

4. Within the new Edit method, check if **ModelState.IsValid** is **false**, reconstruct the view model (add the list of suppliers that did not come back with the request) and send back the view.

```
if (!ModelState.IsValid) {
  var suppliers = _dataContext.Suppliers.ToList();
  pvm.Suppliers = suppliers.Select(s => new SelectListItem
    { Text = s.CompanyName, Value = s.Id.ToString() }).ToList();
  return View(pvm);
}
```

5. After the code from the previous step, construct a product from the data in the view model, save the changes to the database, and redirect the user to the home page.

```
var product = new Product
{
  Id = pvm.Id,
  ProductName = pvm.ProductName,
  UnitPrice = pvm.UnitPrice,
  Package = pvm.Package,
  IsDiscontinued = pvm.IsDiscontinued,
  SupplierId = pvm.SupplierId
};
_dataContext.Attach(product);
_dataContext.Entry(product).State = EntityState.Modified;
_dataContext.SaveChanges();
return RedirectToAction("Index", "Home");
```

> *We are note checking for any database errors that may occur but we will handle that in a future lab.*

6. Run the application and test the product edit functionality.

End of Lab

Lab 13

Objectives

- Add data validation for editing a product

Procedures

1. If not already open, re-open your solution from Lab 12.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 13.*

2. Use what you have learned so far to add some data validation for when the user submits the product edit form.

   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. 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.

End of Lab

Lab 14

Objectives

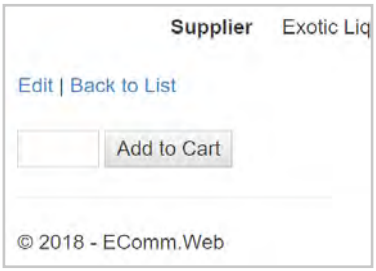
- Implement **shopping cart** functionality (Part 1)

Procedures

1. If not already open, re-open your solution from Lab 13.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 14.*

2. Start be adding a form to the bottom of the product detail page that appears like the screenshot below:




```
<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>
```

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

```
[HttpPost]
public IActionResult AddToCart(int id, int quantity)
{
}
```

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

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]
public IActionResult AddToCart(int id, int quantity)
{
  var product = _dataContext.Products.SingleOrDefault(p => p.Id == 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("_AddedToCartPartial", message);
}
```

> *The next step will be to create the _AddedToCart partial 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 named **_AddedToCartPartial.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 and test the form. You should see the message but as a full-page reload (no menu or footer).

> *We could have easily used a full view that is based on our layout. However, what we really should do is change the view to submit the form asynchronously via ajax and inject the content of the partial view into the existing page.*

7. Add a ** ** element to the bottom of **Detail.cshtml** 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 **Detail.cshtml** named **scripts**.

```
@section scripts {
}
```

> *This section is defined as optional in _Layout.cshtml. So, we can include JavaScript that is specific to this view but know that the layout will decide where in the overall response the script will appear.*

9. Add the JavaScript that will be used to submit the form via **ajax**. This code is available in the **Code/Lab14** folder of the lab bundle if you prefer to copy-and-paste the code.

```
@section scripts {
  <script type="text/javascript">
    $(document).ready(function() {
      $('form').submit(function(event) {
        var formData = {
          'quantity':$('input[name=quantity]').val()
        };
        $.ajax({
          type:'POST',
          url:$('#addToCartForm').attr('action'),
          data:formData
        })
          .done(function(response) {
            $('#message').html(response);
           });
        event.preventDefault();
      });
    });
  </script>
}
```

> *Of course, we could move this JavaScript into a separate file and use the scripts section to include that file.*

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

End of Lab

Lab 15

Objectives

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

Procedures

1. If not already open, re-open your solution from Lab 14.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 15.*

   > *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. This needs to be added **before** the MVC middleware is added.

```
app.UseSession();

app.UseMvc(routes =>
  ...
```

   > *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. Add a new class to the **Models** folder of EComm.Web named **ShoppingCart** with the code necessary to store a collection of **line items** where each line item consists of a **product** and a **quantity**.

```
public class ShoppingCart
{
  public ShoppingCart()
  {
    LineItems = new List<LineItem>();
  }

  public List<LineItem> LineItems { get; set; }
  public string FormattedGrandTotal
  {
    get { return $"{LineItems.Sum(i => i.TotalCost):C}"; }
  }

  public class LineItem
  {
    public Product Product { get; set; }
    public int Quantity { get; set; }
    public decimal TotalCost
    { get { return Product.UnitPrice.Value * Quantity; } }
  }
}
```

> *Since we know that we will be storing an object of this type in session, we should go ahead and define some helper methods that can be used for doing that job.*

5. 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 = JsonConvert.DeserializeObject<ShoppingCart>(json);
  }
  return cart ?? new ShoppingCart();
}
```

> *Notice that we are using an argument of type ISession. This will allow us to use dependency injection to provide for better testability.*

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

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

7. Add code to the **AddToCart** action in **ProductController** that updates the cart. This code should be added immediately before the return statement.

```
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);
```

> *Notice that we are using HttpContext.Session within the action here. It would be nice if we could eliminate this through the use of dependency injection. That will be a bonus task at the end of this lab.*

> *The next step will be to create the "view cart" page.*

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

```
public IActionResult Cart()
{
  var cart = ShoppingCart.GetFromSession(HttpContext.Session);
  return View(cart);
}
```

> *Next, we'll add some markup for the view itself.*

9. Right-click on the **Views/Product** folder, choose **[Add > Existing Item...]**, and select **Cart.cshtml** from the **Code/Lab15** folder of the lab bundle.

10. 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.*

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

```
<ul class="nav navbar-nav">
  <li><a asp-area="" asp-controller="Home" asp-action="Index">...
  <li><a asp-area="" asp-controller="Home" asp-action="About">...
  <li><a asp-area="" asp-controller="Home" asp-action="Contact">...
</ul>
<ul class="nav navbar-nav navbar-right">
  <li><a asp-controller="Product" asp-action="Cart">Shopping Cart</a></li>
</ul>
```

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

13. As a **bonus** task, try to figure out how to eliminate the **hard-coded dependency** on **HttpContext.Session** within ProductController.

End of Lab

Lab 16

Objectives

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

Procedures

1. If not already open, re-open your solution from Lab 15.

   If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 16.

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

2. Add a new class to the **ViewModels** folder named **CartViewModel** with the following properties and attributes.

```
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.

```
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 EComm.Web.ViewModels.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 **_CheckoutFormPartial.cshtml** from the **Code/Lab16** folder of the lab bundle.

6. Take a moment to **examine** the contents of **_CheckoutFormPartial.cshtml**. Take special note of the tag helpers related to **validation**.

7. Modify **Cart.cshtml** to that it includes the **_CheckoutFormPartial** partial view.

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

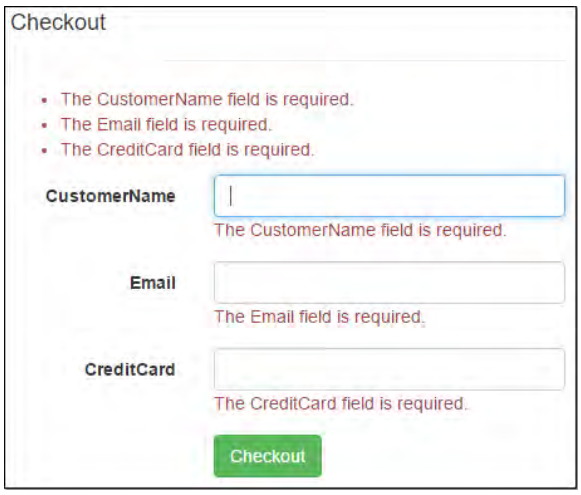
> *In order for client-side validation to function, the jQuery validation JavaScript must be included. A partial view with the necessary files is already part of the project but it is not included in the layout since they will most likely not be needed on every page.*

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 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]
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 would 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 would 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** 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).

> *If you would like to test that server-side validation is running, simply comment-out the inclusion of _ValidationScriptsPartial in Cart.cshtml. You should have the same validation experience but with a post-back occurring before the validation messages appear.*

End of Lab

Lab 17

Objectives

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

Procedures

1. If not already open, re-open your solution from Lab 16.

> *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 17.*

> *In this lab, we will add an admin area to our application. We will use the cookie middleware to implement forms-based authentication and claims-based authorization.*

2. Add the authentication middleware service in **ConfigureServices**.

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddAuthentication(
      CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie();
```

3. Enable the authentication middleware in the **Configure** method. Make sure this is done **before** the call to UseMvc.

```
public void Configure(IApplicationBuilder app, ...)
{
  ...
  app.UseAuthentication();

  app.UseMvc(routes =>
  {
    routes.MapRoute(
      name: "default",
      template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

> *Since we will be using claims-based authorization, we will next configure authorization as a service and add a policy that specifies that a specific claim is required.*

4. Configure **authorization** as a service in the **ConfigureServices** method and add a policy named **AdminsOnly** that requires a role claim of **Admin**.

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

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

```
public class LoginViewModel
{
  [Required]
  public string Username { get; set; }
  [Required]
  public string Password { get; set; }

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

> *Next, we'll add the controller that will handle all authentication-related tasks.*

6. 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.

```
public IActionResult Login(string ReturnUrl)
{
  return View(new LoginViewModel { ReturnUrl = ReturnUrl });
}
```

> *Now that we have an action to return the login view, we'll add the view itself.*

7. Add a new subfolder under **Views** named **Account**.

8. Right-click on the **Views/Account** folder, choose **[Add > Existing Item...]** and select **Login.cshtml** from the **Code/Lab17** folder of the lab bundle.

> *Before we create an action to handle the submission of the login form, let's check to see if the user gets redirected to the login form when they should.*

9. Add an **Authorize** attribute to the **Detail** action of **ProductController**.

```
[Authorize(Policy="AdminsOnly")]
[Route("product/{id:int}")]
public IActionResult Detail(int id)
```

10. **Run** the application and attempt to view the **details** for a product. You should be redirected to the login page.

> *Now that we know we will be sent to the login page, let's add code to do the actual forms-based authentication.*

11. Add a new **asynchronous** action to the **AccountController** named **Login**. This method should have the **HttpPost** and **ValidateAntiForgeryToken** applied and accept a **LoginViewModel**.

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

> *Authenticating against a back-end credential store is oftentimes an expensive operation. Therefore, the sign-in method provided by the framework is asynchronous and our action should also be asynchronous.*

12. 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) return View(lvm);

  var principal = new ClaimsPrincipal(
    new ClaimsIdentity(new List<Claim>
    {
      new Claim(ClaimTypes.Name, lvm.Username),
      new Claim(ClaimTypes.Role, "Admin")
    }, "FormsAuthentication"));

  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.*

13. **Run** the application and test the functionality by trying to view the details of a product. You should be able to login (using "test" and "password") and be redirected back to the product.

14. As a **bonus** exercise, add a item to the menu that a user can use to sign-out.

15. As an **additional bonus** exercise, create an experiment where the user credentials are valid but the claims principal does not contain the required claim for the requested action. Provide a better user experience for this scenario.

End of Lab

Lab 18

Objectives

- Configure **status code pages**

Procedures

1. If not already open, re-open your solution from Lab 17.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 18.*

2. **Run** the application, view a product (you will need to sign-in), and change the URL to make a request for a product that **does not exist** (e.g. /product/9999). The result will be a **404** but the user experience is not the best.

3. Add a new controller named **ErrorController** that will handle returning a response based on the status code.

   > *Our project already has a view model named ErrorViewModel and a view named Error in the Shard folder.*

4. Define an **Index** action that accepts the **status code** as a parameter and returns a view named **Error**.

```
[Route("error/{statusCode:int}")]
public IActionResult Index(int statusCode)
{
  var evm = new ErrorViewModel();
  ViewBag.StatusCode = statusCode;
  return View("Error", evm);
}
```

5. Modify the **<h2>** element in **Error.cshtml** so the **status code** is displayed. Remove the rest of the content after the <h2> element.

```
<h2 class="text-danger">
  A @ViewBag.StatusCode error occurred while processing your request.
</h2>
```

6. Add code to the **Configure** method that configures the **StatusCodePages** middleware to use our new controller action and view. Be sure to add this before the call to UseMvc.

```
app.UseStatusCodePagesWithReExecute("/error/{0}");
```

7. **Run** the application again and repeat the test you performed in **step 2**. You should now see the customized error page.

> *Conditional code could of course be added to the error action so that a different view is returned depending on the value of the status code.*

End of Lab

Lab 19

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 solution from Lab 18.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 19.*

2. Right-click on the test folder in the **EComm** solution, choose **[Add > New Project...]**, select **xUnit Test Project (.NET Core)**, 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. Select **[Test > Windows > Test Explorer]** from the Visual Studio menu and **build** the solution. The test should appear in the Test Explorer window.

5. Click **Run All** in the Test Explorer window and confirm that the test passes.

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

   > *Before we can test code that exists in EComm.Web, we need to add a dependency to EComm.Tests.*

6. Right-click on **Dependencies** under **EComm.Tests** and add a reference for **EComm.Web**, **EComm.DataAccess**, **and EComm.Model**.

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

```
[Fact]
public void ProductDetails()
{
  // Arrange

  // Act

  // Assert
}
```

> *The comments are of course not required but they can provide a good structure to work from.*

> *In the Arrange section, we would like to create an instance of ProductController. However, ProductController requires an instance of DataContext. So, we'll create an in-memory version of DataContext with some hard-coded data (i.e. a stub) and pass that in.*

8. Add the **Microsoft.EntityFrameworkCore.InMemory** NuGet package to the EComm.Tests project.

9. Add a **private method** to **UnitTest1** that creates an **in-memory** version of an **ECommDataContext**. Feel free to use different product values.

```
private ECommDataContext CreateStubContext()
{
  var optionsBuilder = new DbContextOptionsBuilder<ECommDataContext>();
  optionsBuilder.UseInMemoryDatabase("TestDb");
  var context = new ECommDataContext(optionsBuilder.Options);

  // Add sample data
  context.Products.Add(new Product { Id = 1, ProductName = "Milk",
                       UnitPrice = 2.50M });
  context.Products.Add(new Product { Id = 2, ProductName = "Bread",
                       UnitPrice = 3.25M, SupplierId = 1 });
  context.Products.Add(new Product { Id = 3, ProductName = "Juice",
                       UnitPrice = 5.75M });
  context.Suppliers.Add(new Supplier { Id = 1, CompanyName = "Acme" });

  context.SaveChanges();
  return context;
}
```


10. Add code to the **Arrange** section of the **ProductDetail** test that creates a new **ProductController** and passes in an ECommContext **stub**.

```
// Arrange
var controller = new ProductController(CreateStubContext());
```

11. Add code to the **Act** section of the **ProductDetails** test that invokes the **Detail** action. You should see an error. The EComm.Tests project does not know what an **IActionResult** is.

```
// Act
var result = controller.Detail(2);
```

12. Add the **Microsoft.AspNetCore.Mvc.ViewFeatures** NuGet package to the project.

> *Visual Studio may prompt you to add this package when you hover over the error. However, you may still have to add it manually.*

> *Now that we have a result, we can test a variety of things about it. In your applications, what you test for will change depending on what the action does. For this action, we'll check that we got back a ViewResult and we'll check the type and contents of the model.*

13. Add code to the **Assert** section of the **ProductDetails** test that checks the type of the result, the type of the model, and the contents of the model.

```
// 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);
```

14. **Build** the solution.

15. **Run** all the tests and check that they both pass. Feel free to modify the data so that the test will fail and check the behavior.

End of Lab

Lab 20

Objectives

- Build a **Web API** for product data

Procedures

1. If not already open, re-open your solution from Lab 19.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder for Lab 20.*

2. Add a new method to **ProductController** that can be used to obtain data for **all products**.

```
[HttpGet("api/products")]
public IActionResult Get()
{
  var products = _dataContext.Products.ToList();
  return new ObjectResult(products);
}
```

   > *We are choosing to put the Web API method in the same controller that we are using for our views. Sometimes, you may choose to put the Web API methods into a separate controller.*

   > *The use of "api" as part of the route is a common convention but this is not required. You just need to make sure not to use a route that conflicts with another route in your application.*

3. **Run** the application, use Postman to make a GET request for **/api/products**, and examine the response.

4. Add another method to **ProductController** for obtaining a **single product**. Try to implement this yourself before looking at the code on the next page.

```
[HttpGet("api/product/{id:int}")]
public IActionResult Get(int id)
{
  var product = _dataContext.Products.SingleOrDefault(p => p.Id == id);
  if (product == null) return NotFound();

  return new ObjectResult(product);
}
```

5. **Run** the application and use Postman to test the new method.

6. Add another method for **modifying** an **existing product**. Once again, you are encouraged to implement this on your own before looking at the code that follows.

```
[HttpPut("api/product/{id:int}")]
public IActionResult Put(int id, [FromBody]Product product)
{
  if (product == null || product.Id != id) {
    return BadRequest();
  }
  var existing = _dataContext.Products
      .SingleOrDefault(p => p.Id == id);

  if (existing == null) return NotFound();

  existing.ProductName = product.ProductName;
  existing.UnitPrice = product.UnitPrice;
  existing.Package = product.Package;
  existing.IsDiscontinued = product.IsDiscontinued;
  existing.SupplierId = product.SupplierId;
  dataContext.SaveChanges();

  return new NoContentResult();
}
```

> *Your method may look different and that's okay as long as you are meeting all of the requirements (checking for a bad request and not attempting to update a product that doesn't exist).*

7. **Run** the application and test the functionality. An easy way to test this with Postman is to make a GET request for the product, copy the response, paste the JSON into the body for the request and modify it. Don't forget to set the **Content-Type** header to **application/json**.

8. If you have additional time, implement the methods for **create** and **delete**.

End of Lab