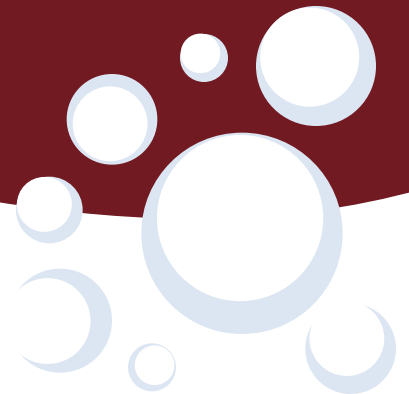


# ASP.NET Core 8 Development Slides





# ASP.NET Core 8 Development

## Agenda

- Introduction
- .NET SDK
- What's New in C#
- ASP.NET Core Application Architecture
- Application Configuration
- Request Routing
- Models
- Controllers
- Views

# ASP.NET Core 8 Development

## Agenda

- HTML Forms
- Input Validation
- Application State
- Error Handling
- Logging
- Testing
- Web APIs
- Security

# ASP.NET Core 8 Development


## Agenda


- gRPC
- Blazor
- Deployment

# ASP.NET Core 8 Development

## Introduction

- Evolution of the .NET Platform
- .NET SDKs and Runtimes
- IDE Choices

- 
- 2002 .NET 1.0, C# 1.0, Visual Studio .NET
  - 2003 .NET 1.1, Visual Studio 2003
  - 2004 Mono 1.0
  - 2005 .NET 2.0, C# 2.0, Visual Studio 2005  
*Generics, Nullable Value Types*
  - 2006 .NET 3.0, Mono 1.2  
*WPF, WCF, WF*
  - 2007 .NET 3.5, C# 3.0, Visual Studio 2008  
*LINQ, Anonymous Types, Lambda Expressions, Extension Methods, Implicit Typing*
  - 2008 Entity Framework 1.0
  - 2009 ASP.NET MVC 1.0
  - 2010 .NET 4.0, C# 4.0, ASP.NET MVC 2, Visual Studio 2010  
*Named / Optional Arguments, Dynamic Binding*
  - 2012 .NET 4.5, C# 5.0, Mono 3.0, ASP.NET MVC 4, Visual Studio 2012  
*Asynchronous Members (async / await)*
  - 2013 .NET 4.5.1, ASP.NET MVC 5, Visual Studio 2013  
*SignalR 1.0*

- 
- 2015 .NET 4.6, C# 6.0, Mono 4.0, Visual Studio 2015, Visual Studio Code 1.0  
*Expression Bodied Members, Null Propagator, String Interpolation*
  - 2016 Xamarin Acquisition, .NET Core 1.0, .NET Standard 1.0  
*Entity Framework Core 1.0*
  - 2017 .NET 4.7, .NET Core 2.0, C# 7.0, Visual Studio 2017  
*ASP.NET Razor Pages, Out Variables, Tuples, Ref Locals and Returns*
  - 2018 GitHub Acquisition, .NET Standard 2.0  
*Blazor Server*
  - 2019 .NET 4.8, .NET Core 3.0, C# 8.0, Visual Studio 2019  
*gRPC, Default Interface Methods, Using Declarations, Nullable Reference Types*
  - 2020 .NET 5, C# 9.0  
*Blazor WebAssembly, Records, Init Only Setters, Top-Level Statements*
  - 2021 .NET 6, C# 10.0, Visual Studio 2022  
*.NET MAUI*
  - 2022 .NET 7, C# 11.0  
*Standard-Term Support (STS) release primarily focused on performance*

2023



.NET 8, C# 12

*Long-Term Support (LTS) release [will be supported until November 2026]  
Primary constructors, collection expressions, runtime improvements,  
preview of .NET Aspire, render modes in Blazor, and more*

## Introduction

### Evolution of the .NET Platform

- .NET 1.0 included ASP.NET Web Forms
  - Had the potential to be cross-platform but was only officially supported on Windows
- Current version of this variant is 4.8 and now referred to as ".NET Framework"
- Will be supported for many years to come (end of support for .NET 3.5 SP1 is October 2028)

# Introduction

## Evolution of the .NET Platform

- The ASP.NET MVC web application framework was introduced in 2009
- Initially presented as an alternative to Web Forms (not a replacement)
- Accompanied by a related framework for building services called Web API

# Introduction

## Evolution of the .NET Platform

- In 2016, Microsoft introduced a new variant of .NET called .NET Core
- Many components were completely rewritten
- Fully supported on Windows, macOS, and Linux
- Included a subset of the functionality provided by .NET Framework
  - Focused on web-based workloads (web UIs and services)
- Merged MVC and Web API into the core framework

## Introduction

### Evolution of the .NET Platform

- The version of .NET Core after 3.1 became the "main line" for .NET and was labeled .NET 5.0
- Supports development of Windows Forms and WPF applications that run on Windows
- The ASP.NET framework in .NET still includes the name "Core" to avoid confusion with previous versions of ASP.NET MVC

## Introduction

### Evolution of the .NET Platform

- The entire .NET platform is made available as open-source
- Community contributions are encouraged via pull requests
  - Thoroughly reviewed and tightly controlled by Microsoft

[github.com/dotnet](https://github.com/dotnet)



# Introduction

## .NET SDKs and Runtimes

- .NET Runtime
  - Different version for each platform
  - Provides assembly loading, garbage collection, JIT compilation of IL code, and other runtime services
  - Includes the dotnet tool for launching applications
- ASP.NET Core Runtime
  - Includes additional packages for running ASP.NET Core applications
  - Reduces the number of packages that you need to deploy with your application

# Introduction

## .NET SDKs and Runtimes

- .NET SDK
  - Includes the .NET runtime for the platform
  - Additional command-line tools for compiling, testing, and publishing applications
  - Contains everything needed to develop .NET applications (with the help of a text editor)

## Introduction

### .NET SDKs and Runtimes

- Each version of .NET has a lifecycle status
  - Standard Term Support (STS) – Includes the latest features with a support period of 18 months
  - Long Term Support (LTS) – Has an extended support period of three years
  - Preview – Not supported for production use
  - Out of support – No longer supported

[dotnet.microsoft.com/download](https://dotnet.microsoft.com/download)

## Introduction

### IDE Choices

- Visual Studio is available for Windows and macOS
  - Full-featured IDE
- Visual Studio Code is available for Windows, macOS, and Linux
  - Includes IntelliSense and debugging features
  - Thousands of extensions are available for additional functionality

[visualstudio.microsoft.com](https://visualstudio.microsoft.com)

## Introduction

### IDE Choices

- JetBrains also offers an IDE for .NET development called Rider
- Available for Windows, macOS, and Linux
- Includes advanced capabilities in the areas of refactoring, unit testing, and low-level debugging

[www.jetbrains.com/rider](http://www.jetbrains.com/rider)

## ASP.NET Core 8 Development

### .NET SDK

- Installation
- Version Management
- Command-Line Interface (CLI)

# .NET SDK

## Installation

- The .NET SDK is distributed using each supported platform's native install mechanism
- Requires administrative privileges to install
- A list of installed SDK versions is available by using the .NET Command Line Interface (CLI)

```
dotnet --list-sdks
```

- A complete list of all installed runtimes and SDKs (as well as the default version) is also available

```
dotnet --info
```

# .NET SDK

## Version Management

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

```
{  
  "sdk": {  
    "version": "6.0.14"  
  }  
}
```

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

# .NET 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
- Include a global.json file in a source control repository to ensure every member of the team is using the same version of the SDK
  - Will generate an error if the specified SDK version is not present on the system

# .NET SDK

## Version Management

- While the SDK version (tooling) is specified using a global.json file, the runtime version is specified within the project file

```
<PropertyGroup>  
  <TargetFramework>net8.0</TargetFramework>  
</PropertyGroup>
```

# .NET SDK

## Version Management

- When an application is launched, it will automatically use the newest available runtime with the same major and minor version number
  - For example, if version 8.0 is specified, the application will use automatically use the 8.0.3 runtime but will not automatically use version 8.1 of the runtime
- Allows for system administrators to apply security patches and runtime bug fixes without the need to recompile and re-deploy the application
- Behavior can be overridden by specifying a RollForward policy value

# .NET SDK

## Version Management

- The target framework for a project can be an older version than the version of the SDK that you are using
  - For example, you can use version 8 of the SDK to build an application that targets the .NET 6 runtime

```
<PropertyGroup>  
  <TargetFramework>net6.0</TargetFramework>  
</PropertyGroup>
```

- Recommended approach – Use the newest version of the tools possible and choose a runtime target based on your deployment environment

# .NET SDK

## Command-Line Interface (CLI)

- 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 SDK

## Command-Line Interface (CLI)

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

## .NET 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 I

### .NET SDK

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



# ASP.NET Core 8 Development

## What's New in C#

- Introduction
- Nullable Reference Types
- Record Types
- Init Only Setters
- Global Using Directives
- File-Scoped Namespace Declarations
- Top-Level Statements
- Primary Constructors
- Collection Expressions

## What's New in C#

### Introduction

- C# 10 introduced with .NET 6
- C# 11 introduced with .NET 7
- C# 12 introduced with .NET 8
- Several new features and improvements
  - Complete list available in the online documentation

## What's New in C#

### Nullable Reference Types

- By default, value types in .NET cannot be set to null
  - A variable can be defined as a nullable value type so that it can store a null value

```
int? num = null;
```

- In the past, reference types could always be set to null and would default to null if not provided with an initial value

```
Product p; // p is null
```

## What's New in C#

### Nullable Reference Types

- The most common exception encountered during .NET development has always been the `NullReferenceException`
  - Occurs when attempting to access the member of an object that is null
- Safety can be significantly improved by using types that cannot be null unless explicitly identified to allow it

# What's New in C#

## Nullable Reference Types

- C# 8 introduced the idea of nullable reference types
  - Like values types, reference types would not be allowed to be null unless the variable was defined as nullable
- Because of the impact on existing code, this feature was not enabled by default
- Could be enabled via the Nullable annotation in the project file

```
<Nullable>enable</Nullable>
```

- Starting with .NET 6, nullable reference types were enabled by default

# What's New in C#

## Nullable Reference Types

- If enabled, compiler warnings will be generated when...
  - Setting a non-nullable reference type to null
  - Defining a reference type that does not initialize all non-nullable reference type members as part of construction
  - Dereferencing a possible null reference without checking for null (or using the null-forgiving operator)

```
string fn = person!.FirstName;
```

## What's New in C#

### Nullable Reference Types

- It is a good idea to enable nullable reference types for new projects
- Refactoring an existing application to use nullable reference types could require a significant amount of effort

## What's New in C#

### Record Types

- Every type in .NET is either a value type or a reference type
  - Struct is a value type
  - Class is a reference type
- Value types are recommended to be defined as immutable and are copied on assignment
  - Use value semantics for equality
  - Supports additional safety and optimizations especially for concurrent programming with shared data

# What's New in C#

## Record Types

- The record type introduced in C# 9 allows you to easily define an immutable reference type that supports value semantics for equality

```
public record Person
{
    public string LastName { get; }
    public string FirstName { get; }

    public Person(string first, string last) =>
        (FirstName, LastName) = (first, last);
}
```

# What's New in C#

## Record Types

- None of the properties of a record can be modified once it's created
- Records do support inheritance
- It is easy to create a new record from an existing one via the with keyword

```
var person = new Person("Joe", "Smith");
Person brother = person with { FirstName = "Bill" };
```

## What's New in C#

### Record Types

- Record types can be a very good fit for things like ViewModels and Data Transfer Objects (DTOs)

## What's New in C#

### Init Only Setters

- It is very convenient to initialize the properties of an object by using object initialization syntax

```
var product = new Product { Name = "Bread", Price = 2.50 }
```

- However, in the past, this was only possible by defining the properties as writable

## What's New in C#

### Init Only Setters

- In C# 9, it is now possible to define properties with init only setters
- Properties can be set as part of object initialization but become read-only after that

```
public class Product
{
    public string Name { get; init; }
    public double Price { get; init; }
}
```

## What's New in C#

### Global Using Directives

- C# 10 introduced global using directive support
- If the global keyword is present, the using directive will be in effect for every file in the project

```
global using EComm.Core;
```

- Can be in any file but a good practice is to have a separate cs file for all the project's global using directives

## What's New in C#

### Global Using Directives

- Starting with .NET 6, global using directives for common system namespaces can be included implicitly via a project setting

```
<ImplicitUsings>enable</ImplicitUsings>
```

- This setting is included in new projects by default
- For an ASP.NET project, there are a total of 16 namespaces that will be implicitly referenced

## What's New in C#

### File-Scoped Namespace Declarations

- Typically, code within a namespace is defined within curly braces

```
namespace Acme.Models  
{  
    ...  
}
```

- Starting with C# 10, a namespace declaration can be used to specify that all the code within a file belongs to a namespace

```
namespace Acme.Models;  
...  

```



# What's New in C#

## Top-Level Statements

- A .NET application requires an entry point function named Main defined within a static class

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

- The C# compiler (10 and later) can recognize executable code that is outside of a class as the code for the entry point and generate the necessary function and static class for you

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

# What's New in C#

## Top-Level Statements

- ASP.NET project templates combine the implicit using feature with top-level statements to minimize the amount of code required in Program.cs

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

# What's New in C#

## Primary Constructors

- Starting with C# 12, you can add parameters to the declaration of a struct or class

```
public class Person(string firstName, string lastName)
{
    //
}
```

- This will cause the compiler to create a primary constructor that initializes generated private fields

```
var p = new Person("Bill", "Gates");
```

```
public string FullName => $"{firstName} {lastName}";
```

# What's New in C#

## Collection Expressions

- Collection expressions provide an easier way to initialize a collection

```
int[] a = [1, 2, 3, 4, 5, 6, 7, 8];
```

```
List<string> b = ["one", "two", "three"];
```

- Using a spread element in a collection expression adds all the elements in that expression

```
int[] a = [1, 2, 3];
int[] b = [4, 5, 6];
int[] c = [.. a, .. b];    // results in [1, 2, 3, 4, 5, 6]
```

# ASP.NET Core 8 Development

## Application Architecture

- Introduction
- NuGet Packages
- Application Startup
- Hosting Environments
- Middleware and the Request Pipeline
- Services and Dependency Injection

## Application Architecture

### Introduction

- Single stack for Web UI and Web APIs
- Modular architecture distributed as NuGet packages
- Flexible, environment-based configuration
- Built-in dependency injection support
- Support for using an MVC-based architecture or a more page-focused architecture by using Razor Pages
- Blazor allows for the implementation of client-side functionality using .NET code

# Application Architecture

## NuGet Packages

- NuGet is a package manager for .NET
  - [www.nuget.org](http://www.nuget.org)
- All the libraries that make up .NET (and many 3<sup>rd</sup>-party libraries) are distributed as NuGet packages
- NuGet package dependencies are stored in the project file

```
<PackageReference Include="Microsoft.EntityFrameworkCore" Version="8.0.6" />
```

# Application Architecture

## NuGet Packages

- The dotnet restore command will fetch any referenced NuGet packages that are not available locally
- Uses nuget.org as the package source by default
- Additional or alternative package sources (remote or local) can be specified by using a nuget.config file

# Application Architecture

## NuGet Metapackages

- Metapackages are a NuGet convention for describing a set of packages that are meaningful together
- Every .NET Core project implicitly references the Microsoft.NETCore.App package
  - ASP.NET Core projects also reference the Microsoft.AspNetCore.App package
- These two metapackages are included as part of the runtime package store
  - Available anywhere the runtime is installed

# Application Architecture

## Application Startup

- When an ASP.NET Core application is launched, the first code executed is the application's Main method
  - Generated by the compiler if using top-level statements
- Code in the Main method is used to...
  - Create a WebApplication object
  - Configure application services
  - Configure the request processing pipeline
  - Run the application

# Application Architecture

## Application Startup

- WebApplication's CreateBuilder method is typically used to create the WebApplicationBuilder object
- When the WebApplicationBuilder is created, it loads configuration information from...
  - appsettings.json and appsettings.{Environment}.json
  - User secrets (when running in Development environment)
  - Environment variables and command-line arguments

# Application Architecture

## Application Startup

- After the WebApplicationBuilder has been initialized, application services can be added
- WebApplicationBuilder's Build method is used to construct the WebApplication object and initialize the dependency injection system
- The WebApplication object is used to configure the request processing pipeline

# Application Architecture

## Application Startup

- A collection of framework services are automatically registered with the dependency injection system
  - IHostApplicationLifetime
    - Used to handle post-startup and graceful shutdown tasks
  - IHostEnvironment / IWebHostEnvironment
    - Has many useful properties (ex. EnvironmentName)
  - ILoggerFactory
  - IServer
  - And many others...

# Application Architecture

## Application Startup

- The environment for local machine development can be set in the launchSettings.json file
  - Overrides values set in the system environment
  - Only used on the local development machine
  - Is not deployed
  - Can contain multiple profiles

# Application Architecture

## Application Startup

- By default, clients can use HTTP/2 when selected during the TLS handshake; otherwise, HTTP/1.1 is used
- Additional configuration options are described in the documentation

# Application Architecture

## Hosting Environments

- EnvironmentName property can be set to any value
- Framework-defined values include:
  - Development
  - Staging
  - Production (default if none specified)
- Typically set using the ASPNETCORE\_ENVIRONMENT environment variable
- Can also be configured via launchSettings.json



# Application Architecture

## Middleware

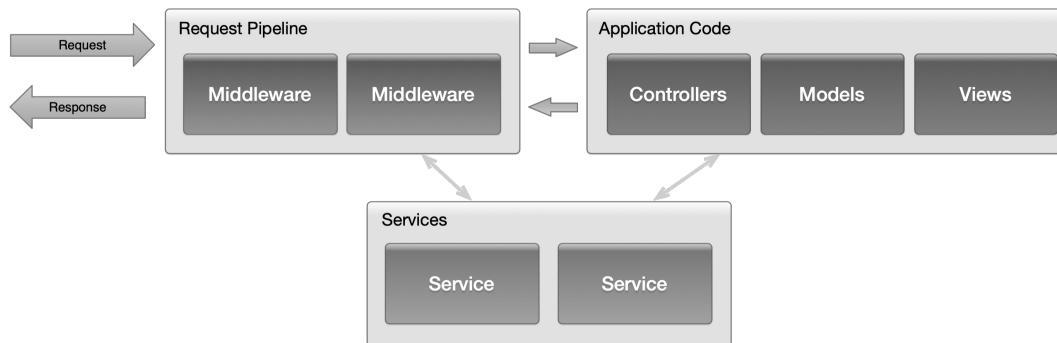
- ASP.NET uses a modular request processing pipeline
- The pipeline is composed of middleware components
- Each middleware component is responsible for invoking the next component in the pipeline or short-circuiting the chain
- Examples of middleware include...
  - Request routing
  - Handling of static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

# 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

# Application Architecture



# Application Architecture

## Pipeline

- The last piece of middleware in the pipeline is typically the routing middleware
- Routes the incoming request to a controller
- Instead of using controllers, the new minimal API framework in .NET can be used

## Lab 2

### Application Architecture

- Create a new ASP.NET Core 8 web project in Visual Studio
- Run and test the application

## ASP.NET Core 8 Development

### Application Configuration

- Middleware
- Services
- Configuration Providers and Sources
- Configuration API
- Options Pattern

# Application Configuration

## Middleware

- 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"

```
app.UseHttpsRedirection();  
app.UseAuthorization();
```

- The order in which middleware is added to the pipeline can be important
  - Determines the order of execution
  - As an example, it would be very important for authentication middleware to execute before some caching middleware that could return a cached response

# Application Configuration

## Services

- 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

## Services

- Services are typically added via extension methods available on IServiceCollection

```
builder.Services.AddDbContext<ApplicationDbContext>(...);  
builder.Services.AddScoped<IEmailSender, MyEmailSender>();  
builder.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

## Services

- Services are available throughout the application via dependency injection
- A common practice is to follow the Explicit Dependencies Principle
  - Controllers include all required services as constructor parameters
  - System will provide an instance or throw an exception if the type cannot be resolved via the DI system

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

# Application Configuration

## Services

- If using controllers (and not the minimal API feature), some services must be added during startup

```
builder.Services.AddControllers();
```

- Other methods are available if using views or razor pages

# Application Configuration

## Configuration Providers and Sources

- Before ASP.NET Core, application settings were typically stored in an application's web.config file
- ASP.NET Core introduced 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 WebApplicationBuilder adds providers to read settings (in the order shown) from:
  - appsettings.json
  - appsettings.{Environment}.json
  - User secrets
  - System environment variables
  - Command-line arguments
- Values read later override ones read earlier

# 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 : Controller
{
    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"
  }
}
```

```
public class HomeController : Controller
{
    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; }
}
```

```
builder.Services.Configure<EmailOptions>(Configuration.GetSection("Email"));
```

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

# ASP.NET Core 8 Development

## Request Routing

- RESTful Services
- Endpoint Routing
- Route Attributes
- Route Templates
- Route Constraints
- Route Template Precedence

# Request Routing

## RESTful Services

- When configuring request routing, you should try to maintain a RESTful API
- Clean, extension-less URLs that identify resources
- Use of the correct HTTP verbs within an API
- 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

## Endpoint Routing

- Routing is responsible for mapping request URLs to endpoints and dispatching incoming requests to those endpoints
- Routing can also be used to generate URLs that map to endpoints
  - Eliminates hardcoded URLs that would need to be updated when the routing configuration changes

## Request Routing

### Endpoint Routing

- A collection of extension methods on `ApplicationBuilder` are available for adding different types of endpoints
  - All start with the word "Map"
- Verb-based methods make it easy to configure simple endpoints
  - `MapGet`, `MapPost`, `MapPut`, `MapDelete`, etc.
- `MapControllers` will configure and add an endpoint for each controller action defined in the application

## Request Routing

### Route Attributes

- If using controllers, attributes can be used to define the routing information used to construct the endpoints
- The `Route` attribute will create an endpoint for all HTTP verbs

```
[Route("products")]  
public IActionResult AllProducts() { ... }
```

- Verb-specific attributes should be used to define an endpoint for a specific HTTP verb

```
[HttpGet("products")]  
public IActionResult AllProducts() { ... }
```

# Request Routing

## Route Attributes

- A controller-level attribute can be used to specify a prefix for all the actions of the controller

```
[Route("[controller]")]
public class ProductController : Controller
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id) { ... }
}
```

- In the example above, a request for a product would use a URL of <https://example.com/product/6>

# Request Routing

## Route Templates

- 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/{name}/{id}`

`{language}-{region}/library/{topic}`



`{language}{region}/{topic}`

## 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

```
products/{sort=price}
```

- 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)

```
product/{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 a 404 (resource not found)
- Invalid input should typically result in a different response (e.g., 400 with an appropriate error message)

# Request Routing

## Route Template Precedence

- Each route template is assigned a value by the system based on how specific it is
- Literal segments are considered more specific than parameter segments

`/hello/customer`

`/hello/{name}`

- A parameter segment with a constraint is considered more specific than one without a constraint
- The Order property of an endpoint can be used to override the default precedence behavior
- If a request matches multiple endpoints with the same precedence, an `AmbiguousMatchException` is thrown at runtime

# ASP.NET Core 8 Development

## Models

- Introduction
- Persistence Ignorance
- Dependency Inversion
- Clean Architecture
- Asynchronous Data Access
- Object-Relational Mapping
- Entity Framework Core
- Dapper ORM

## Lab 3

### Models

- Create a database with some sample data



# Models

## Introduction

- Models represent "real world" objects the user is interacting with
- Entities are the objects used during Object-Relational Mapping and provide a way to obtain and persist model data
- The term Data Transfer Object (DTO) is often used to describe an object that carries data between different processes or subsystems
  - A single DTO may contain multiple different entities, exclude some entity properties, or use different property names
  - In a Web API application, the object that get serialized into JSON is often a DTO

# Models

## Persistence Ignorance

- The model data typically comes from an external source (database, web service, file, etc.)
- For better maintainability and testability, it is a best practice to use a data access component to encapsulate the details about where the model data comes from
- In ASP.NET, data access should be performed by a service made available via dependency injection
  - Makes it easy to test components independently with hard-coded data (no database)

# Models

## Dependency Inversion

- One of the SOLID design principles is Dependency Inversion
- "High-level modules should not depend on low-level models. Both should depend on abstractions."
  - The web application/service should not be built based on how the data access library was designed
  - An interface should be used to define the functionality that the web application requires
  - The data access library should provide a component that implements the required interface

# Models

## Clean Architecture

- To help facilitate reuse and a clear separation of concerns, we will apply some principles from a design philosophy known as clean architecture
- Our models (business/domain objects) and data access interface (abstraction) will be defined in a core library
- Our technology-specific data access implementation will be defined in an infrastructure library

# Models

## Clean Architecture

- The extra work necessary to achieve clean separation will provide more flexibility when it comes to...
  - Unit testing
  - Comparing different data access technologies
  - Switching to a different persistence mechanism (e.g., microservices)

# Models

## Asynchronous Data Access

- When performing IO-bound operations (database access, web service calls, etc.), it is a best practice to perform that work asynchronously
- Allows for the efficient use of thread resources
  - Thread pool threads can be used to handle other incoming requests while the IO-bound operation is in progress
  - Improves the scalability of a web application

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

## Lab 4

### Models

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

## 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 by with ADO.NET, or several frameworks exist that can help with this task
  - Entity Framework Core
  - Dapper (3<sup>rd</sup>-party micro-ORM)
  - AutoMapper (mapping one object to another)

# Models

## Entity Framework Core

- 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

- EF Core 8 was released alongside .NET 8
- Last few releases of EF include many internal performance improvements
  - Less database round trips for when SaveChanges only includes a single change (up to 45% speed improvement)
  - Getting the key for a newly inserted record uses OUTPUT instead of SELECT (eliminates a server-side operation)
  - Improvements for batch operations and lazy loading

# Models

## Entity Framework Core

- EF Core 7 and 8 also added some new features
  - JSON column mapping
  - Mapping entities to stored procedures (ability that exists in EF6 for .NET Framework)
  - Value generation on value converters used for key properties
  - New interceptors for low-level actions
  - Support for database-specific aggregate functions
  - Table per Concrete Type (TPC) mapping
  - Entity splitting

```
modelBuilder.Entity<Person>()  
    .SplitToTable("PeopleLastNames", s => s.Property(p => p.LastName));
```

# Models

## Entity Framework Core

- By creating a subclass of DbContext, EF Core can populate your entity 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 entity 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 entities 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 sends the SQL it generates to the logging system when executed
- Interception API can also be used to obtain or modify the SQL

```
public class HintCommandInterceptor : DbCommandInterceptor
{
    public override InterceptionResult ReaderExecuting(DbCommand command,
        CommandEventData eventData, InterceptionResult result)
    {
        {
            command.CommandText += " OPTION (OPTIMIZE FOR UNKNOWN)";
            return result;
        }
    }
}
```

```
services.AddDbContext(b => b.UseSqlServer(connStr)
    .AddInterceptors(new HintCommandInterceptor()));
```

## Models

### Entity Framework Core

- EF Core can also be used to execute custom SQL or call a stored procedure

```
var products = context.Products
    .FromSqlRaw("SELECT * FROM dbo.Products")
    .ToList();
```

```
var product = context.Products
    .FromSqlRaw("EXECUTE dbo.GetProduct {0}", id)
    .SingleOrDefault();
```

- In the example above, EF Core uses an ADO.NET parameterized query and SQL injection is not a concern
  - Still an issue if the entire string is constructed first and then passed to FromSqlRaw

# Models

## Entity Framework Core

- The entity classes can be defined manually, or the code for them can be automatically generated
  - CLI tools
  - Package Manager Console tools in Visual Studio

# Models

## Entity Framework Core

- EF Core is a large topic and in-depth coverage is beyond the scope of this course
  - Inheritance
  - Shadow Properties
  - Cascading Updates and Deletes
  - Transactions
  - Concurrency Conflicts
  - Migrations

[docs.microsoft.com/en-us/ef/core/](https://docs.microsoft.com/en-us/ef/core/)

## Models

### Dapper ORM

- Dapper is an open-source ORM framework that has become a very popular alternative to Entity Framework

`github.com/StackExchange/Dapper`

- Has less features than EF but provides a good high-performance "middle-ground" between ADO.NET and EF

## Lab 5

### Data Access

- Create an EF-based implementation of IRepository

# ASP.NET Core 8 Development

## Controllers

- Responsibilities
- Requirements and Conventions
- Dependencies
- Action Results

## Controllers

### Responsibilities

- The action executed for a particular endpoint is typically a method of a controller
- A controller may need to retrieve or make modifications to model data
- The controller also often determines the appropriate type of response to return
  - HTML, JSON, XML, redirection, error, etc.

# Controllers

## Responsibilities

- Controller methods that are reachable via the routing system are referred to as controller actions
- 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 a system class called Controller (or its subclass ControllerBase for an API)
    - 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 : Controller
{
    private IEmailSender _emailSender;

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

# Controllers

## Action Results

- IActionResult is implemented by a variety of different return types
- Framework uses the ExecuteResultAsync when creating the HTTP response

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

- Writing directly to the response should be avoided
  - Adds a dependency to the HTTP context
  - Make things like unit testing more difficult

# 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 NotFound();`
  - 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

```
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());
}
```

## Controllers

### Action Results

- API controllers will typically return an entity type or a DTO

```
[HttpGet("products")]  
public IEnumerable<Product> GetAllProducts()
```

- System will create an `ActionResult` and look at the incoming request to support content negotiation
- `ActionResult` is still useful when an action can return different return types

```
public IActionResult GetProduct(int id)  
{  
    if (!_repository.TryGetProduct(id, out var product)) {  
        return NotFound();  
    }  
    return Ok(product);  
}
```

## Controllers

### Asynchronous Controller Actions

- It is common for a controller action to invoke an asynchronous method to perform an IO-bound operation
  - Database access, web service call, etc.
- The action should be marked as `async` with a return type of `Task<T>` and `await` used with the asynchronous method

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



# Controllers

## Asynchronous Controller Actions

- Making an action asynchronous does not change the experience for the client
  - No response is sent until the entire action is complete
- Can improve application scalability by allowing the thread pool thread to handle other incoming requests while waiting for the IO-bound operation to complete
- It is also possible to accept a CancellationToken that can be used to handle the cancellation of a long-running request

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

## Lab 6

### Controllers

- Register a service that returns an IRepository
- Modify a controller to accept a dependency
- Return a response that includes database data

# ASP.NET Core 8 Development

## Views (Part I)

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

## Views

### Responsibilities

- 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 define views that contain a mix of HTML and code
  - View files are transformed into C# classes that implement the `IView` interface
  - Razor Pages use the same syntax

# Views

## Conventions

- It is a common convention to place all views in a folder at the root of the project named Views
  - Razor pages use a folder called Pages
- 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

- 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
  - Helps to prevent many cross-site scripting (XSS) style attacks
- Can be disabled by using `Html.Raw( )`
  - Should only be used for trusted content

```
<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>100 * 0.10</span>
```

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

```
<span>10</span>
```

## Views

### Razor Syntax

- To render @ into the response, 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

- The Controller base class defines a property called ViewData that is a key-value store of type <string, object>
- The ViewData object is passed to the ViewResult when it is created by the View( ) method

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

- The ViewData object is accessible within the view

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

# Views

## View Data

- ViewBag is a property that acts as a wrapper around ViewData
- Defined as type dynamic so it can be used to access ViewData without the use of brackets and quotes
- Does not provide any additional features other than a cleaner syntax
- ViewData and ViewBag can be used interchangeably

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

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

```
public ActionResult Index()
{
    ViewBag.Message = "Hello";
    return View();
}
```

```
<h1>@ViewData["Message"]</h1>
```

# Views

## View Data

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

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

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

# 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.Entities.Product>
```

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

```
@using EComm.Entities
```

# 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 can be passed to the partial view with the for attribute

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

## Lab 7

### Views

- Use a partial view to display the list of products

## ASP.NET Core 8 Development

### Views (Part II)

- HTML and URL Helpers
- Tag Helpers
- View Components
- Hands-On Lab Exercise
- Client-Side Dependencies
- Razor Pages
- View Models

## 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 properties

## 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 HTML helper, define a new extension method for `IHtmlHelper`
- 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 introduced 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 inherits from `ViewComponent` and implements an `InvokeAsync` method

```
public async Task<IViewComponentResult> InvokeAsync()
{
    var products = await _repository.GetAllProducts();
    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 if 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 / namespace
- When returning a view, the paths searched include
  - `/Views/{Controller Name}/Components/{View Component Name}/{View Name}`
  - `/Views/Shared/Components/{View Component Name}/{View Name}`
  - `/Pages/Shared/Components/{View Component Name}/{View Name}`
- The default view name for a view component is `Default`

```
return View();
```

# Views

## View Components

- In EComm.Web, we are using a partial view for the product list
- Since the partial view is used on the home page, HomeController must fetch the list of products and pass it to the view that hosts the partial view
  - This should not be a responsibility of HomeController
  - If we would like the product list to appear somewhere else in the application, that controller would also need to fetch the products
- If the product list is a view component, it can be self-contained (fetch its own data)

# Lab 9

## Views

- Refactor the product list to be a view component

# Views

## Client-Side Dependencies

- When creating a modern web application, you often end up using one or more client-side frameworks
  - 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 3<sup>rd</sup>-party client-side dependencies
  - Webpack, npm, and Yarn are some examples
- An option provided by Microsoft is the Microsoft Library Manager (LibMan)
  - [github.com/aspnet/LibraryManager](https://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@3.6.0",
      "destination": "wwwroot/lib/jquery",
      "files": [
        "jquery.min.js"
      ]
    }
  ]
}
```

## Views

### Client-Side Dependencies

- ASP.NET Core views can conditionally include client-side files based on environment name
- A version string can 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

### Razor Pages

- Razor Pages were introduced in ASP.NET Core 2
- Makes coding page-focused scenarios easier and more productive
- Razor Page support is added to an application in a manner similar to MVC support

```
services.AddRazorPages();
```

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



# Views

## Razor Pages

- Including the @page directive in a razor page file creates an endpoint for that page based on its name
  - 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 PrivacyModel
@{
    ViewData["Title"] = "Privacy Policy";
}
<h1>@ViewData["Title"]</h1>

<p>Use this page to detail your site's privacy policy.</p>
```

# Views

## Razor Pages

- A code-behind file (e.g., Privacy.cshtml.cs) is typically used to define a class that will act as the controller and model for the page

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

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

# 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 an entity directly to a view
- Every view should be strongly-typed to a view model type
- This can provide some benefits but requires more effort

# 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

# ASP.NET Core 8 Development

## HTML Forms

- Introduction
- Form Tag Helper
- Input Tag Helper
- Select Tag Helper
- Hands-On Lab Exercise
- Form Submissions
- Model Binding
- Hands-On Lab Exercise

# 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
  - Generates a hidden request verification token to help prevent cross-site request forgery (CSRF)

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

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

## HTML Forms

### Input Tag Helper

- The Input Tag Helper can be used to display textual model data

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

- Generates the id and name HTML attributes
- Sets the HTML type attribute value
- Generates HTML5 validation attributes from data annotations

## HTML Forms

### Select Tag Helper

- The Select Tag Helper can be used to generate an HTML select element and associated option elements

```
<select asp-for="SupplierId" asp-items="Suppliers"></select>
```

- The property used for asp-items should be a collection of type SelectListItem
  - Best for a view model to provide this rather than the view (logic in the view) or the entity (web specific type in the entity)

## Lab 10

### 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

```
[HttpPost("product/edit/{id}")]  
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("product/edit/{id}")]  
[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

## HTML Forms

### Saving Changes with EF Core

- EF Core has a change tracking system
- A call to `SaveChanges` (or `SaveChangesAsync`) will cause EF to submit the necessary SQL to persist all pending changes
- An entity can be attached to a `DbContext` and then marked as modified

```
Products.Attach(product);  
Entry(product).State = EntityState.Modified;  
await SaveChangesAsync();
```

- Requires only one trip to the database but does update all columns of the entity

## HTML Forms

### Saving Changes with EF Core

- If the entity is fetched via EF first and then modified, EF can determine which properties are different and then update only those columns

```
var product = Products.Single(p => p.Id == id);  
product.ProductName = "Something Else";  
product.UnitPrice = 15.99;  
// set other properties  
await SaveChangesAsync();
```

- Update query is more efficient but requires two trips to the database

## Lab 11

### HTML Forms

- Complete the ability to edit a product

## ASP.NET Core 8 Development

### Input Validation

- Introduction
- Data Annotations
- Model Binding
- Input Tag Helpers
- Validation Tag Helpers
- Hands-On Lab Exercise

# Input 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

# Input 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 used 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; }
}
```

# Input 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

# Input Validation

## Input Tag Helpers

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

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

- If any validation attributes are present, the helper will inject HTML attributes to enable client-side data validation

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

- The message displayed on the client can be customized by providing a value for the ErrorMessage parameter of the validation attribute

# Input 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 used by the helpers in the view when returning a view after a server-side validation error

# Input 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>
```

# Input 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 12

### Input Validation

- Add data validation for when editing a product

# ASP.NET Core 8 Development

## Application State

- Introduction
- HttpContext.Items
- Session State
- TempData
- Hands-On Lab Exercises

## 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 can add something to Items so that it 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.AddMemoryCache();  
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

## Labs 13 – 15

### Shopping Cart

- Implement shopping cart functionality
- These three labs are optional depending on the experience and interest of the group
  - They can be completed in full, demonstrated by the instructor, done after the course, or skipped entirely
  - None of the future labs require these labs to be completed

## Lab 13

### Shopping Cart (Part I)

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

## Lab 14

### Shopping Cart (Part 2)

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

## Lab 15

### Shopping Cart (Part 3)

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

# ASP.NET Core 8 Development

## Error Handling

- Best Practices
- HTTP Error Status Codes
- Status Code Pages
- Developer Exception Page
- Hands-On Lab Exercise

## 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
- By default, ASP.NET Core returns an empty response body for client error status codes

## Error Handling

### Status Code Pages

- The StatusCodePages middleware can be used to define the response body that should be returned for client error status codes

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

- By default, this middleware will return a simple string describing the error

## Error Handling

### Status Code Pages

- The StatusCodePages middleware can also...
  - Use a custom function to provide a response

```
app.UseStatusCodePages(async context =>  
    await context.HttpContext.Response.SendAsync($"Status code:  
    {context.HttpContext.Response.StatusCode}", "text/plain");
```

- Redirect (302) the user to a different page

```
app.UseStatusCodePagesWithRedirects("/clienterror?code={0}");
```

- Return the HTTP error status code but with the results from a different action (URL in browser does not change)

```
app.UseStatusCodePagesWithReExecute("/clienterror", "?code={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 a development environment

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



## Error Handling

### Exception Handling

- If an exception occurs during initial application startup...
  - The hosting layer logs a critical exception
  - The dotnet process crashes
- If running on IIS or IIS Express, the ASP.NET Core Module will return a 502.5 – Process Failure response for an application that is unable to start

## Error Handling

### Exception Handling

- When not using the developer exception page, the Exception Handling Middleware should be used

```
if (env.IsDevelopment()) {  
    app.UseDeveloperExceptionPage();  
}  
else {  
    app.UseExceptionHandler("/error");  
}
```

- MVC template includes a default action, view model, and view
- Razor Pages template includes a default error page

## Error Handling

### Exception Handling

- Use `IExceptionHandlerPathFeature` to access the exception and original request path

```
var exceptionHandlerPathFeature =  
    HttpContext.Features.Get<IExceptionHandlerPathFeature>();  
  
if (exceptionHandlerPathFeature?.Error is FileNotFoundException)  
    ExceptionMessage = "File error thrown";  
  
if (exceptionHandlerPathFeature?.Path == "/index")  
    ExceptionMessage += " from home page";
```

## Lab 16

### Error Handling

- Examine the default error behavior
- Add a controller for centralized error handling
- Add and configure the `StatusCodePages` middleware
- Modify the `ExceptionHandler` middleware

# ASP.NET Core 8 Development

## Logging

- Introduction
- Configuration
- ILogger
- Serilog and Seq

## 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 3<sup>rd</sup>-party logging frameworks can be used to provide many other logging options
  - Serilog
  - NLog
  - Log4Net
  - Loggr
  - elmah.io

# Logging

## Configuration

- The list of logging providers can be configured when configuring the host

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

# Logging

## ILogger

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

```
public class ProductController : Controller
{
    public ProductController(IRepository repository,
        ILogger<ProductController> logger) { }
}
```

# Logging

## ILogger

- `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);
```

# Logging

## ILogger

- The highest verbosity level written to the log is typically set in appsettings

```
"Logging": {  
  "LogLevel": {  
    "Default": "Debug",  
    "System": "Information",  
    "Microsoft": "Information"  
  }  
}
```

# Logging

## Serilog

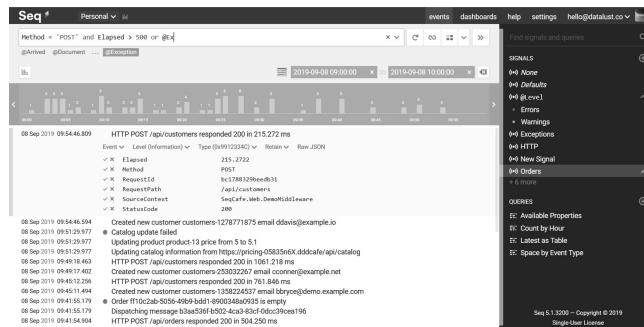
- Serilog has become a popular choice for ASP.NET Core
  - Wide variety of destinations and formats
  - Can record structured event data

[github.com/serilog/serilog-aspnetcore](https://github.com/serilog/serilog-aspnetcore)

# Logging

## Seq

- In many ASP.NET Core applications, the log data needs to be off-host and centralized (e.g., load-balanced environment)
- Seq is an open-source server that can accept logs via HTTP
  - Integrates with .NET Core, Java, Node.js, Python, Ruby, Go, Docker, and more



© Treeloo, Inc. - All rights reserved (24-169)

233

# Logging

## Maintainability

- One important note is that the logging framework(s) that you choose should not change how you write to the log (ILogger)
  - The only code that changes is in Program.cs

© Treeloo, Inc. - All rights reserved (24-169)

234

# ASP.NET Core 8 Development

## Testing

- Introduction
- Unit Testing
- xUnit
- Testing Controllers
- Hands-On Lab Exercise
- 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 inherent 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 the mock object is being used by the object under test

# 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 project templates for the MSTest, xUnit, and NUnit frameworks
- xUnit has steadily been gaining in popularity both inside and outside of Microsoft

# Testing

## xUnit

- Fact attribute is used to define a test that represents something that should always be true
- Theory attribute is used to define a test that represents something that should be true for a particular set of data

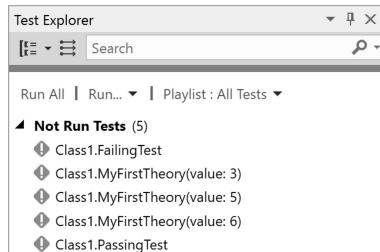
```
[Fact]
public void PassingTest()
{
    Assert.Equal(4, Add(2, 2));
}
```

```
[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?

## Lab 17

### Unit 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

## 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

- The Microsoft.AspNetCore.Mvc.Testing package provides a collection of components to help with integration testing
  - Test web host
  - In-memory test server
  - WebApplicationFactory

# Testing

## Integration Testing

- ASP.NET Core's WebApplicationFactory class can be used to create a host for the application

```
public class ECommAppFactory<TStartup> : WebApplicationFactory<TStartup> where TStartup: class
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            // configure app services here like in the app's
            // ConfigureServices method
        })
    }
}
```

# Testing

## Integration Testing

- Test classes can use the custom WebApplicationFactory to create an HttpClient

```
public class PageTests :  
    IClassFixture<ECommApplicationFactory<EComm.Web.Startup>>  
{  
    private readonly HttpClient _client;  
    private readonly CustomWebApplicationFactory<EComm.Web.Startup> _factory;  
  
    public PageTests(CustomWebApplicationFactory<EComm.Web.Startup> factory)  
    {  
        _factory = factory;  
        _client = factory.CreateClient(new WebApplicationFactoryClientOptions  
        {  
            AllowAutoRedirect = false  
        });  
    }  
}
```

# Testing

## Integration Testing

- Your HttpClient can issue the same HTTP requests that a browser would and receive the same HTTP responses

```
[Theory]  
[InlineData("/")]  
[InlineData("/Index")]  
[InlineData("/About")]  
public async Task Get_EndpointsSuccessAndContentType(string url)  
{  
    // 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());  
}
```



# ASP.NET Core 8 Development

## Web APIs

- API Controllers
- OpenAPI / Swagger
- Testing APIs
- Retrieval Operations
- Model Binding
- Update, Create, and Delete Operations
- Cross-Origin Request Sharing (CORS)

## Web APIs

### API Controllers

- ASP.NET Core includes a class named ControllerBase
  - Includes many properties and methods for handling HTTP requests
- The Controller class inherits from ControllerBase and adds support for views
- If creating a controller that does not have any views, you should inherit directly from ControllerBase

## Web APIs

### API Controllers

- An API controller should be decorated with the ApiController attribute

```
[ApiController]  
public class ProductApiController : ControllerBase
```

- Automatic HTTP 400 responses for validation failures
- Problem details for error status codes

## Web APIs

### OpenAPI / Swagger

- OpenAPI is a specification for describing REST APIs
- Swagger is a collection of tools that work with OpenAPI
  - SwaggerDocument objects expose data about the API in JSON format (openapi.json)
  - Swagger UI is a dynamically generated web-based UI that can be used to view and test API methods

## Web APIs

### OpenAPI / Swagger

- There are two main OpenAPI implementations for .NET
  - Swashbuckle
  - NSwag
- Both include an embedded version of Swagger UI
  - Made available via middleware

## Web APIs

### OpenAPI / Swagger

- By default, the API project templates include a reference to Swashbuckle.AspNetCore
- SwaggerDocument generation is handled by a service

```
services.AddSwaggerGen();
```

- Document availability and Swagger UI is configured via middleware components

```
app.UseSwagger();  
app.UseSwaggerUI();
```

## Web APIs

### OpenAPI / Swagger

- The ProducesResponseType attribute should be used when defining Web API actions

```
[HttpPost]  
[ProducesResponseType(StatusCodes.Status201Created)]  
[ProducesResponseType(StatusCodes.Status400BadRequest)]  
public ActionResult<Product> Create(Product product)
```

- Used by tools like Swagger to generate more descriptive documentation

## Web APIs

### OpenAPI / Swagger

- Actions (or entire controllers) can be omitted from the Swagger document generation process by using the ApiExplorerSettings attribute

```
[ApiExplorerSettings(IgnoreApi = true)]  
public class ErrorController : Controller
```

## Web APIs

### Testing APIs

- API endpoints that are exposed via GET are easy to test using a web browser
- For other verbs, it can be helpful to have a tool that can be used to craft custom HTTP requests
  - Postman application is very popular (getpostman.com)
  - Many other options are available

## Web APIs

### Testing APIs

- Microsoft recently introduced a new tool called the HTTP REPL

```
dotnet tool install -g Microsoft.dotnet-httprepl
```

- Command-line tool for making HTTP requests
- Supports most of the HTTP verbs
- Can use Swagger documents to discover the endpoints

```
> https://localhost:5001/~ ls
Products  [get|post]
Customers [get|post]
```

[docs.microsoft.com/en-us/aspnet/core/  
web-api/http-repl](https://docs.microsoft.com/en-us/aspnet/core/web-api/http-repl)

# 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

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

# Web APIs

## Retrieval Operations

- There are several options available for altering the format of the JSON returned
  - Attributes
  - Custom formatter
  - Data projection

```
public async Task<IActionResult> GetProduct(int id)
{
    var product = await _repository.GetProduct(id, true);
    if (product == null) return NotFound();
    var retVal = new {
        Id = product.Id, Name = product.ProductName,
        Price = product.UnitPrice,
        Supplier = product.Supplier.CompanyName
    };
    return Ok(retVal);
}
```

## Lab 18

### Web API

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

## Web APIs

### Update Operations

- In a Web API, update operations are performed with...
  - HTTP PUT – Replaces an existing resource
  - HTTP PATCH – Modifies part of an existing resource
- If successful, the response should be HTTP 204 (no content)

```
public async Task<IActionResult> PutProduct(int id, Product product)
{
    if (id != product.Id) return BadRequest();

    var existingProduct = await _repository.GetProduct(id);
    if (existingProduct == null) return NotFound();

    await _repository.SaveProduct(product);
    return NoContent();
}
```

## 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 resource
- The `CreatedAtAction` and `CreatedAtRoute` methods can be used to generate a correctly formatted response

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

## 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();
```



## Lab 19

### Web API

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

## 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("https://example.com"));
```

# ASP.NET Core 8 Development

## Security

- Authentication
- ASP.NET Core Identity
- Authorization
- Cookie-Based Authentication
- Web API Authentication
- Duende IdentityServer
- Secrets Management

## Security

### Authentication

- Authentication is the process of discovering the identity of an entity through an identifier and verifying the identity by validating the credentials provided by the entity
- It is common to validate the credentials once and generate a temporary unique token that is used to authorize other actions
  - The token can contain other information (claims) about the entity including group membership and privileges
  - Care must be taken to ensure the token is sent in a secure manner and the integrity of the token can be verified

# Security

## Authentication

- Authentication in ASP.NET Core is handled by the `IAuthenticationService`
  - Uses registered authentication handlers (`IAuthenticationHandler`)
  - Triggered by authentication middleware

# Security

## Authentication

- A configured authentication handler is also called a scheme
  - Registered as a service using `AddAuthentication`
  - Multiple schemes can be registered but a default scheme must be identified, or the scheme specified in the `Authorize` attribute

```
services.AddAuthentication(CookieAuthenticationDefaults.  
    AuthenticationScheme).AddCookie();
```

## Security

### Authentication

- The RemoteAuthenticationHandler type is used for authentication that requires a remote authentication step
- Uses a callback to complete the authentication process
- Used for OAuth 2.0 and OpenID Connect (OIDC) based authentication
  - Facebook, Twitter, Google, Microsoft, etc.

## Security

### Authentication

- The authentication middleware is added by calling UseAuthentication in the Configure method
- If a request is not authenticated, authorization invokes a challenge using the default (or specified) authentication scheme
- Authentication challenge examples:
  - Redirection to a login page for a Web UI
  - 401 response with www-authentication: bearer header for a Web API

## Security

### Authentication

- If a request is authenticated but the action is not permitted for the authenticated user, a forbid action is triggered
- Forbid action examples:
  - Redirection to a page explaining the situation
  - 403 response

## Security

### ASP.NET Core Identity

- ASP.NET Core Identity is a complete, full-featured authentication provider for creating and managing logins
- 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
- There is extensive documentation and several example projects

`docs.microsoft.com/en-us/aspnet/core/  
security/authentication/identity`

## Security

### 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() { ... }
}
```

## Security

### 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() { ... }
```

## Security

### Authorization

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

```
public IActionResult Delete(int id)
{
    if (User.HasClaim("SomeClaim")) {
        //
    }
}
```

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

## Security

### Cookie-Based Authentication

- 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
  - Makes it easy to create an authentication system with a custom UI and custom credential store
  - Saves you from having to reimplement the low-level components

## Security

### Cookie-Based Authentication

- First step is to configure the authentication services

```
services.AddAuthentication(CookieAuthenticationDefaults.  
    AuthenticationScheme).AddCookie();
```

- as well as the authentication and authorization middleware

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

## Security

### Cookie-Based Authentication

- An object of type `CookieAuthenticationOptions` can be passed to the call to `AddCookie( )`
  - `LoginPath` (route to the login page)
  - `LogoutPath`
  - `SlidingExpiration`
  - and many more...



## Security

### Cookie-Based Authentication

- When using the cookie middleware, it is your responsibility to create the login page and process the submission
- You are free to do anything necessary to authenticate the user
  - Access a database
  - Call a web service
  - Query Active Directory

## Security

### Cookie-Based Authentication

- Once authenticated, construct a ClaimsPrincipal which will be used to construct the authentication cookie
- The ClaimsPrincipal contains a ClaimsIdentity that contains a 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, "bill@microsoft.com"),  
        new Claim("FullName", "Bill Gates"),  
        new Claim(ClaimTypes.Role, "Admin"),  
    },  
    CookieAuthenticationDefaults.AuthenticationScheme));
```

# Security

## Cookie-Based Authentication

- 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);
```

## Lab 20

### Cookie-Based Authentication

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

## Security

### Web API Authentication

- For API authentication, cookie-based authentication is typically not used, and it is not possible to directly present a UI to gather credentials
- In some scenarios, a simple form of authentication where the credentials are passed with each request is sufficient
  - HTTP Basic
  - Pre-shared key (PSK) authentication
- In a microservice architecture, an API Gateway can be used to perform authentication and proxy requests to other services

## Security

### Web API Authentication

- A modern flexible approach to API authentication is to use bearer token authentication
  - ASP.NET Core supports OAuth 2.0 and OpenID Connect

```
services.AddAuthentication(options => {
    options.DefaultAuthenticateScheme =.AspNetCore.Authentication
        .JwtBearer.JwtBearerDefaults.AuthenticationScheme;

    options.DefaultChallengeScheme =.AspNetCore.Authentication
        .JwtBearer.JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options => {
    options.Authority = identityUrl;
    options.Audience = "EComm";
});
```

## Security

### Web API Authentication

- Authority property is the address of the token-issuing authentication server
  - Used by the middleware to obtain the public key for validation of the token signature
- Audience property represents the resource the token grants access to
  - Value must match the parameter in the token

## Security

### Web API Authentication

- The token issuing server can be an external identity provider (Facebook, Twitter, Microsoft, etc.) or you can create your own using ASP.NET Core
- If issuing your own security tokens, it is recommended to use a third-party library to handle many of the security-related details
  - IdentityServer4
  - OpenIddict

# Security

## Web API Authentication

- It is also possible to create a custom authentication scheme / handler
  - Can create a subclass `AuthenticationHandler` and override `HandleAuthenticateAsync`
- When developing a custom authentication scheme, extreme care should be taken to ensure security vulnerabilities are not introduced

## Lab 21

### API Authentication

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

## Security

### Duende IdentityServer

- IdentityServer is a standards-compliant OpenID Connect and OAuth 2.0 framework for ASP.NET Core
- OAuth2 is a protocol that allows applications to request access tokens from a security token service
  - Allows authentication and authorization to be centralized
- OpenID Connect is an extension on top of OAuth2
  - Concrete implementation of OAuth2 standards

## Security

### Duende IdentityServer

- Starter, Business, and Enterprise Editions
- Community Edition is a free version that is equivalent to the Enterprise Edition
  - Must meet certain requirements (gross revenue, etc.)

## Security

### Duende IdentityServer

- IdentityServer provides middleware that adds endpoints to an arbitrary ASP.NET Core host
- Project templates are available to provide a starting point

```
dotnet new install Duende.IdentityServer.Templates
```

## Security

### Duende IdentityServer

- Exercise: Protecting an API using Client Credentials

# Security

## Secrets Management

- Secrets are configuration values that are in some way sensitive
  - Connection strings
  - API keys
- It is a good practice for secrets not to be in source code or config files (things that end up in source control)
- Secrets should be made available to the production environment through a controlled means
  - Environment variables
  - Azure Key Vault
  - AWS Secrets Manager

# Security

## Secrets Management

- For local development, the Secret Manager tool can be used
- Secrets are stored in a JSON configuration file in a system-protected user profile folder on the local machine
- Use the init command or select Manage User Secrets in Visual Studio to enable user secrets for a project

```
dotnet user-secrets init
```

- Adds an entry into the project file

```
<PropertyGroup>  
  <TargetFramework>netcoreapp3.1</TargetFramework>  
  <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>  
</PropertyGroup>
```



# Security

## Secrets Management

- Secrets can be added via the command-line or by using the Manage User Secrets command in Visual Studio

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345"
```

- CreateDefaultBuilder calls AddUserSecrets when the EnvironmentName is Development

# ASP.NET Core 8 Development

## gRPC

- Introduction
- Protobuf
- Server
- Client
- Limitations

# gRPC

## Introduction

- gRPC is a language-agnostic, high-performance Remote Procedure Call (RPC) framework
  - Contract-first API development
  - Tooling available for many languages to generate strongly-typed clients
  - Support for streaming on the client and server
- In-depth coverage of gRPC is beyond the scope

# gRPC

## Protobuf

- By default, gRPC uses Protocol Buffers (protobuf)
- High-performance binary serialization format
- Services are defined in .proto files that are present on the server and client

```
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

# gRPC

## Protobuf

- Code generation is provided by the Grpc.Tools NuGet package
- .NET Core will then generate the necessary .NET types based on .proto files included in the project file

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" />
</ItemGroup>
```

# gRPC

## Server

- To create the server, inherit from the base class generated by the tooling and override the relevant methods
  - If a method is not overridden, the server will return an HTTP 501 – Not Implemented response

```
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request,
        ServerCallContext context)
    {
        return Task.FromResult(new HelloReply {
            Message = "Hello " + request.Name
        });
    }
}
```

# gRPC

## Server

- An endpoint must be created to expose the service

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

# gRPC

## Client

- To call a gRPC service, a channel is required

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
```

- A client can then be created (uses HttpClient under-the-covers)

```
var client = new ECommGrpc.ECommGrpcClient(channel);
```

- Calls can then be made using the strongly-typed client

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

# gRPC

## Limitations

- gRPC depends on features implemented by HTTP/2
- Service cannot be called by JavaScript in the browser
  - Because of some HTTP/2 features that are not yet supported
- Will not work if a proxy exists between the client and server that does not fully support HTTP/2
- Does work very well as a means of high-performance communication between back-end services

`docs.microsoft.com/en-us/  
aspnet/core/grpc`

# ASP.NET Core 8 Development

## Blazor

- Introduction
- Razor Components
- Blazor Server
- Blazor WebAssembly

# Blazor

## Introduction

- Blazor is a framework for building interactive client-side web UI with .NET code
- Provides an alternative to frameworks such as Angular and React
- Can be used without .NET on the server-side
  - Server just needs to deliver the runtime, assemblies, and static resources to the client

# Blazor

## Razor Components

- Blazor apps are based on razor components
  - UI element such as a dialog or data entry form
  - Written using the same Razor syntax used by traditional views and Razor Pages
- Components render into an in-memory representation of the browser's Document Object Model (DOM) called a render tree
  - Used to determine what updates should be applied to the UI

# Blazor

## Razor Components

```
@page "/counter"
<h1>Counter</h1>
<p>Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">
    Click me
</button>

@code {
    int currentCount = 0;

    void IncrementCount()
    {
        currentCount++;
    }
}
```

# Blazor

## Razor Components

- Razor Components can be used in with different render modes
  - Blazor Server
  - Blazor WebAssembly

# Blazor

## Blazor Server

- With Blazor Server, a JavaScript file (blazor.server.js) is sent to the browser
  - Establishes a persistent connection between the browser and the server using SignalR (WebSockets)
- Message is sent to the server when events occur in the browser
- C# code in the Razor Component is executed on the server
- UI updates are sent to the browser over the SignalR connection
- JavaScript interop allows for...
  - JavaScript functions to be invoked from C#
  - C# code to be triggered from JavaScript

# Blazor

## Blazor Server

- Blazor Server does maintain state information for each connected user
- Scalability of the application is therefore limited by available memory on the server
- Scalability varies based on the application, but Microsoft has tested this and found...
  - 5,000 concurrent users on instance with 1 vCPU and 3.5 GB of memory
  - 20,000 concurrent users on instance with 4 vCPU and 14 GB of memory



# Blazor

## Blazor WebAssembly

- Blazor WebAssembly (WASM) has been getting a lot of attention since its announcement
- Uses the same Razor Components as Blazor Server but the C# code executes on the client
- Microsoft has created a version of the .NET Runtime (CLR) that is written in WebAssembly
  - Allows .NET assemblies (that contain IL code) to be downloaded and executed on the client
  - Code executes in the same browser sandbox as JavaScript
  - Does not require the user to install something ahead of time (browser extension or plug-in)

# Blazor

## Blazor WebAssembly

- The user must be using a browser that supports WebAssembly
  - Firefox 52 (March 2017)
  - Chrome 57 (March 2017)
  - Edge 16 (September 2017)
  - Safari 11 (September 2017)

# Blazor

## Blazor WebAssembly

- When visiting a site that uses Blazor WebAssembly, a small JavaScript file is downloaded (blazor.webassembly.js) that...
  - Downloads the WebAssembly version of the .NET runtime (about 1 MB in size), the app, and all dependencies
- No state is maintained on the server
- .NET classes like HttpClient can be used to communicate with the server
  - Preconfigured in a new Blazor WebAssembly project

# Blazor

## Blazor WebAssembly

- In .NET 8, each Razor component can be configured to use a specific render mode
  - Static Server – Not interactive
  - Interactive Server – Blazor Server
  - Interactive WebAssembly – Blazor WebAssembly
  - Interactive Auto – Blazor Server initially and then switches to Blazor WebAssembly after the necessary resources have been downloaded

## Lab 22

### Blazor

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

## ASP.NET Core 8 Development

### Deployment

- dotnet publish
- Kestrel
- IIS
- Docker

# Deployment

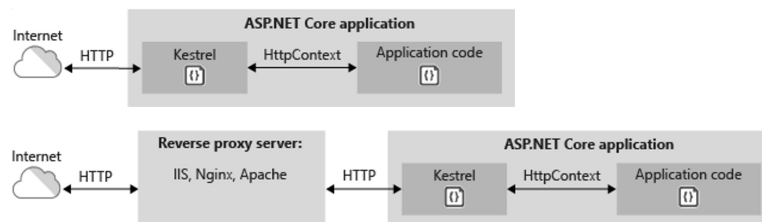
## dotnet publish

- The dotnet publish command compiles an app and copies the files required for deployment into a publish folder
  - Used by Visual Studio's [ Build > Publish ] wizard
- When performing a publish, the app can be...
  - Framework-dependent
    - Does not include the .NET runtime – correct version must already be present on the deployment machine
  - Self-contained
    - Does include the .NET runtime
    - Must choose the target architecture at publish time

# Deployment

## Kestrel

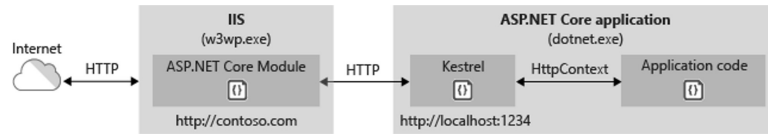
- The Kestrel web server is supported on all platforms where .NET is supported
- You can use Kestrel by itself or with a reverse proxy server



# Deployment

## IIS

- Out-of-process hosting



- In-process hosting



# Deployment

## Documentation

- There is extensive documentation for a wide variety of deployment scenarios
  - Apache httpd, Nginx, etc.

[docs.microsoft.com/en-us/aspnet/core/  
host-and-deploy](https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy)

# Deployment

## Docker

- Docker is an open platform that enables developers and administrators to build images, ship, and run applications in a loosely isolated environment called a container

[www.docker.com](http://www.docker.com)

# Deployment

## Docker

- Developer – "Helps me to eliminate the 'works on my machine' problem"
  - Application is executed and debugged in the same container that is deployed to production
- Administrator – "Allows me to treat hardware instances less like 'pets' and more like 'cattle'"
  - Hardware resources come-and-go with minimal configuration requirements – just need the ability to run a container

# Deployment

## Docker

- 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

# Deployment

## Docker

- 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
  - [hub.docker.com/\\_/microsoft-dotnet-sdk](https://hub.docker.com/_/microsoft-dotnet-sdk)
  - [hub.docker.com/\\_/microsoft-dotnet-aspnet](https://hub.docker.com/_/microsoft-dotnet-aspnet)

# Deployment

## Docker

- 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 -d -p 80:80 ecomm/api
```

- The Microsoft images are preconfigured (via an environment variable) to run Kestrel on port 80
- Documentation has extensive information about configuring Kestrel (HTTPS, HTTP/2, host filtering, etc.)

# Deployment

## Docker

- Visual Studio 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 ]



# Deployment

## Docker

- Docker containers can run on any machine with the Docker engine installed
- 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