Comprehensive Blazor Server

Agenda

- Introduction
- What's New in C#
- Application Architecture
- Introduction to Blazor
- Blazor Server Application
- Data Binding
- Models
- Razor Component Forms
- Custom Components

Comprehensive Blazor Server

Agenda

- Blazorise
- Security and Identity
- JavaScript Interop
- Testing

Comprehensive Blazor Server

Introduction

- Evolution of the .NET Platform
- .NET SDKs and Runtimes
- Visual Studio and Visual Studio Code

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

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

- 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`

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
  - Current – Includes the latest features and bug fixes but will only be supported for a short time after the next release
  - LTS (Long-Term Support) – Has an extended support period
  - Preview – Not supported for production use
  - Out of support – No longer supported

`dotnet.microsoft.com/download`

Introduction

Visual Studio and Visual Studio Code

- 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`

Introduction

Visual Studio and Visual Studio Code

- 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`

Comprehensive Blazor Server

What's New in C#

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

What's New in C#

Introduction

- C# 9 introduced with .NET 5
- C# 10 introduced with .NET 6
- Several new features and improvements
  - Complete list available in the online documentation

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 introduces 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

- In .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
{
  ...
}
```

- C# 10 allows for a namespace declaration to specify that all the code within a file belongs to a specific 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# 10 compiler 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 Core 6 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#

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;
```

- Reference types can store null and 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 is 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 are not allowed to be null unless the variable is 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>
```

- In .NET 6 project templates, this is now included 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
- Values 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)

Comprehensive Blazor Server

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

Application Architecture

NuGet Packages

- NuGet is a package manager for .NET
  - `www.nuget.org`
- All the libraries that make up .NET 6 (and many 3rd-party libraries) are distributed as NuGet packages
- NuGet package dependencies are stored in the project file

```
<PackageReference Include="Microsoft.EntityFrameworkCore" Version="6.0.0" />
```


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

- 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

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

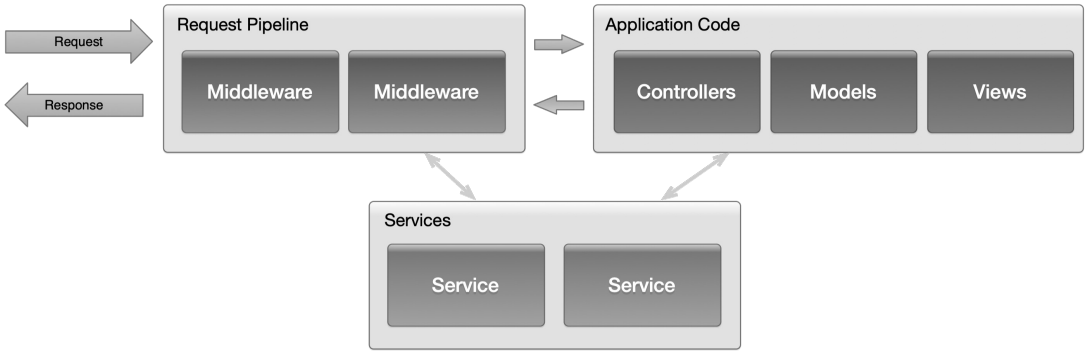
- ASP.NET uses a modular request processing pipeline
- The pipeline is composed of middleware components
- Each middleware component is responsible for invoking the next component in the pipeline or short-circuiting the chain
- Examples of middleware include…
  - Request routing
  - Handling of static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



Comprehensive Blazor Server

Introduction to Blazor

- Overview
- Components
- Hosting Models
- Blazor Server
- Blazor WebAssembly
- Supported Platforms

Introduction to Blazor

Overview

- Blazor is a framework for building interactive client-side web UI with .NET code
- Provides an alternative to frameworks such as Angular and React

Introduction to Blazor

Overview

- Blazor apps are based on 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

Introduction to Blazor

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++;
  }
}
```

Introduction to Blazor

Hosting Models

- Razor Components can be used in one of two different configurations
  - Blazor WebAssembly
  - Blazor Server

Introduction to Blazor

Blazor WebAssembly

- 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)

Introduction to 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

Introduction to Blazor

Supported Platforms

- 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)

Introduction to 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

Introduction to 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
  - Each circuit uses approximately 250 KB of memory for a minimal Hello World-style app

Introduction to Blazor

Blazor Server

- State associated with each client is referred to as a circuit
- Circuits are not tied to a specific network connection
- Makes it possible to tolerate temporary network interruptions and attempts by the client to reconnect to the server

Introduction to Blazor

Blazor Server

- Each browser screen (window or tab) requires a separate circuit
- Closing a tab or navigating to an external URL is a graceful termination
  - Server resources are released immediately
- For a non-graceful termination, resources are retained for a configurable length of time

Introduction to Blazor

Blazor Server vs. Blazor WebAssembly

- Blazor Server
  - Complete .NET API compatibility
  - Direct access to server and network resources
  - Fast initial load time
- Blazor WebAssembly
  - Offline capability
  - Static site hosting
  - Access to native client capabilities
  - Low latency interactions

Comprehensive Blazor Server

Blazor Server Application

- Hosting Blazor Server
- Client-Side Routing
- Layout Components
- Configuration
- Dependency Injection
- Logging
- Error Handling

Blazor Server Application

Hosting Blazor Server

- Blazor Server uses ASP.NET Core SignalR to communicate with the browser
- Blazor works best when using WebSockets as the SignalR transport
- Long Polling is used by SignalR when WebSockets isn't available
  - Typically caused by a VPN or proxy blocking the connection
  - Blazor Server emits a warning if it detects Long Polling must be used

Blazor Application Component

Client-Side Routing

- A single HTML page is initially loaded by the browser
  - Loads the framework components and the App component
- App component includes the client-side Router component
  - Intercepts relative URL requests and loads the component with a matching route (@page directive)
  - <base> element must be modified if the host page is not located at the root of the site

Blazor Application Component

Client-Side Routing

- Components that can be loaded via the routing system (and include a @page directive) are typically placed in the Pages folder
- Components that are used by other components but cannot be navigated to directly are typically placed in the Shared folder

Blazor Application Component

Layout Components

- App component typically specifies a default layout
  - Defines common content (e.g., navigation menu)
- Other component content is loaded into a location specified by the layout (@Body)

Blazor Server Application

Configuration

- 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

Blazor Server Application

Configuration

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

Blazor Server Application

Configuration

- 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

Blazor Server Application

Configuration

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

Blazor Server Application

Configuration

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

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

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


Blazor Server Application

Configuration

- 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;
}
```

Application Configuration

Dependency Injection

- Services are components that are available throughout an application via dependency injection
- The lifetime of a service for a traditional ASP.NET Core application 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

Dependency Injection

- 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

Dependency Injection

- 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 : ControllerBase
{
  public ProductController(IEmailSender emailSender) {
    ...
  }
}
```


Application Configuration

Dependency Injection

- Razor components typically use the @inject directive

```
@page "/fetchdata"

<PageTitle>Weather forecast</PageTitle>

@using MyBlazorServerApp.Data
@inject WeatherForecastService ForecastService
```

Application Configuration

Dependency Injection

- A service can receive other services via DI
- When DI creates the service, it recognizes the services it requires in the constructor and provides them

```
using System.Net.Http;

public class DataAccess : IDataAccess
{
    public DataAccess(HttpClient http)
    {
        ...
    }
}
```


Application Configuration

Dependency Injection

- In Blazor Server apps, the concept of a request is different
- Scoped services aren't reconstructed when navigating among components on the client
  - Communication takes place over the SignalR connection of the user's circuit, not via HTTP requests
- Transient service instances can live longer than expected since components can be long-lived

Application Configuration

Dependency Injection

- Care must be taken when using a service instance that isn't thread safe and designed for concurrent use
  - Prominent example is Entity Framework's DbContext
- A good approach is to use one instance per operation
- Register a factory object as a service
  - Use the factory to obtain an instance when needed

```
@inject IDbContextFactory<ECommContext> DbFactory;

private async Task DeleteContactAsync()
{
  using var context = DbFactory.CreateDbContext();
  ...
```


Application Configuration

Logging

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

Application Configuration

Logging

- The list of logging providers can be configured when configuring the host
- 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) { }
}
```

```
@page "/counter"
@inject ILogger<Counter> logger
```


Application Configuration

Logging

- 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);
```

Application Configuration

Logging

- The highest verbosity level written to the log is typically set in via application configuration

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


Application Configuration

Logging

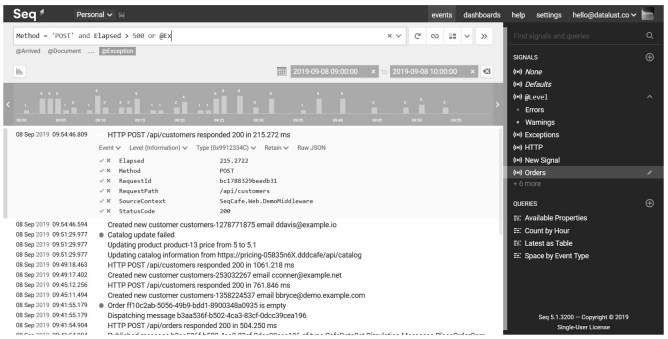
- Serilog has become a popular choice for ASP.NET Core
  - Wide variety of destinations and formats
  - Can record structured event data

github.com/serilog/serilog-aspnetcore

Application Configuration

Logging

- 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



Application Configuration

Logging

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

Application Configuration

Error Handling

- When an error occurs, Blazor apps display a yellow bar at the bottom of the screen
- In a Blazor Server app, customize the experience in the Pages/_Layout.cshtml file

```
<div id="blazor-error-ui">
  <environment include="Staging,Production">
    An error has occurred...
  </environment>
  <environment include="Development">
    An unhandled exception has occurred...
  </environment>
  <a href="" class="reload">Reload</a>
  <a class="dismiss">✕</a>
</div>
```


Application Configuration

Error Handling

- Client-side errors don't include the call stack and don't provide detail on the cause of the error
- During development, circuit error information can be made available to the client by enabling detailed errors

```
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.AspNetCore.SignalR": "Debug"
    }
  }
}
```

Application Configuration

Error Handling

- Blazor treats most unhandled exceptions as fatal to the circuit where they occur
  - Could otherwise leave the circuit in an undefined state
- Will require the user to reload the page to create a new circuit

Application Configuration

Error Handling

- For Blazor apps, you cannot use a middleware component in the request pipeline for global error handling
- Error boundaries provide a convenient approach for handling exceptions
  - Renders its child content when an error hasn't occurred
  - Renders error UI when an unhandled exception is thrown

```
<ErrorBoundary>
  <ChildContent>
    @Body
  </ChildContent>
  <ErrorContent>
    <p class="errorUI">Nothing to see here right now. Sorry!</p>
  </ErrorContent>
</ErrorBoundary>
```

Comprehensive Blazor Server

Data Binding

- Razor
- One-Way Data Binding
- Attribute Binding
- Event Handling
- Two-Way Data Binding
- Default Actions
- Change Detection

Data Binding

Razor

- Components use Razor to define a mix of HTML markup and C# code
- @code section can be used within a Razor file or code can be placed in a separate file by using a partial class
- Fields, properties, and methods defied in C# are available to be used within the markup
- Properties of a component can be set by another component if the [Parameter] attribute is used

Data Binding

One-Way Data Binding

- Data that flows from the component to the DOM
- Events in the DOM that trigger code to execute

```
<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
  int currentCount = 0;

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


Data Binding

Attribute Binding

- The value of an HTML element attribute can also be set with data binding

```
<span class="@BackgroundColor">@currentCount</span>
```

- If bound to a Boolean expression, Blazor will hide the attribute if the value evaluates to false

```
<button class="btn btn-primary"
        disabled="@(currentCount > 10)"
        @onclick="IncrementCount">
```

Data Binding

Event Handling

- Event handlers use the @on<event> syntax
- Can use a member method or a lambda function

```
@onclick="@(() => currentCount++)">Click me</button>
```

- Invocation will cause Blazor to update the UI with the latest values
- An event handler can accept An EventArgs type
  - Specific type depends on the event

Data Binding

Two-Way Data Binding

- DOM will be updated when the component value changes
- Component value will also be updated when the DOM changes
- Could be accomplished by using two one-way bindings

```
<input value="@increment"
       @onchange="@((ChangeEventArgs e)
                    => increment = int.Parse($"{e.Value}"))" />
```

Data Binding

Two-Way Data Binding

- The @bind syntax can be used to achieve two-way binding
- Tied to the onchange event by default
- A different triggering event can be specified

```
<input @bind="@increment" @bind:event="oninput" />
```


Data Binding

Default Actions

- When attaching to an event in Blazor, the browser's default action will still trigger
- This can be explicitly (or conditionally) prevented

```
<input @bind="@increment"
       @onkeypress="KeyHandler"
       @onkeypress:preventDefault />
```

```
<input @bind="@increment"
       @onkeypress="KeyHandler"
       @onkeypress:preventDefault="@shouldPreventDefault" />
```

Data Binding

Change Detection

- When a DOM event triggers code, Blazor assumes values may have been modified
  - Automatically refreshes the UI
- Sometimes, you will need to explicitly tell Blazor to refresh the UI
  - An example is when a background thread updates a field
- Call the StateHasChangegd method of the Component base class

```
var timer = new System.Threading.Timer(
  callback: _ => { IncrementCount(); StateHasChanged(); },
```


Comprehensive Blazor Server

Models

- Introduction
- Asynchronous Data Access
- Object-Relational Mapping
- Entity Framework Core
- Dapper ORM

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

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();
}
```

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 (3rd-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

- 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/`

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
- Dapper is not specifically covered in this course

Comprehensive Blazor Server

Razor Component Forms

- Route Parameters
- NavigationManager
- Accepting User Input
- Validation

Razor Component Forms

Route Parameters

- When a component needs information about what to display, a route parameter can be used to set a component parameter

```
@page "/edit/{id}"
```

```
@code {
  [Parameter]
  public string? Id { get; set; }
}
```

Razor Component Forms

Route Parameters

- A route parameter can be marked as optional
- You can provide a default value in OnInitilaized

```
@page "/welcome/{name?}"
```

```
@code {
  [Parameter]
  public string? Name { get; set; }

  protected override void OnInitialized()
  {
    Name = Name ?? "Guest";
  }
}
```


Razor Component Forms

NavigationManager

- The Blazor Router component automatically handles standard navigation elements (e.g., links)
- When you need to handle navigation tasks programmatically, the NavigationManager object provides helpful events and methods
  - Available to a component via dependency injection

```
@inject NavigationManager NavigationManager
```

```
private void EditProduct(int id)
{
  NavigationManager.NavigateTo($"edit/{id}");
}
```

Razor Component Forms

Accepting User Input

- When creating a component for accepting user input, standard HTML <input> elements are typically used
  - @bind attribute used to connect value to component field
- An HTML <form> element does not need to be used
  - onclick binding on a button can trigger client-side code to make navigation decisions
  - However, using the EditForm component can make validation tasks easier

Razor Component Forms

Validation

- You can implement client-side validation in a manual way via events and data binding
- You can also use the EditForm component bound to a model that uses data annotations
  - Required, EmailAddress, MaxLength, RegularExpression, ...
- DataAnnotationsValidator component can be used to provide user feedback

```
<EditForm Model="@product" OnValidSubmit="SaveProduct">
<DataAnnotationsValidator />
```

Razor Component Forms

Validation

- The ValidationMessage component can be used to display the text of the error message from a data annotation

```
<InputText id="ProductName" @bind-Value="@product.ProductName" />
<ValidationMessage For="@(() => product.ProductName)" />
```

- Visual styles are defined via CSS

```
.valid.modified:not([type=checkbox]) {
    outline: 1px solid #26b050;
}
.invalid {
    outline: 1px solid red;
}
.validation-message {
    color: red;
}
```


Razor Component Forms

Validation

- The EditForm provides callbacks for handling form submissions
  - OnValidSubmit
  - OnInvalidSubmit
  - OnSubmit

Comprehensive Blazor Server

Custom Components

- Introduction
- Child Content
- Two-Way Data Binding Between Components
- Cascading Parameters

Custom Components

Introduction

- It is very common to create reusable shared custom components in a Blazor app
- Components in the Shared folder can easily be used by other components
  - Do not include a @page directive

Custom Components

Child Content

- A custom component may be used to wrap content that is provided to it
- This is commonly done by providing a component parameter named ChildContent of type RenderFragment

Custom Components

Two-Way Data Binding Between Components

- A property of a component can be bound to the property of a child component with @bind-[property]

```
<Alert @bind-Show="ShowAlert">
```

- Child component must implement a property that the parent can use to listen for changes
  - Must use the naming convention of [property]Changed
  - Can be of type Action<propertyType>
  - Invoke the action when the property changes

Custom Components

Two-Way Data Binding Between Components

- Remember that a component will automatically re-render itself when an event handler is triggered but not when a bound property changes in another way (async operation)

```
StateHasChanged();
```

- Using an EventCallback<T> instead of an Action<T> will cause Blazor to see a property change as an event
  - Will invoke StateHasChanged

Custom Components

Cascading Parameters

- Data binding makes it easy for a parent component to pass data to a child component
- Simple data binding can be cumbersome when a parent needs to make data available to a more deeply nested component
- The CascadingValue component can be used to wrap a component hierarchy and supply a value to all the components within its subtree

```
<CascadingValue Value="@Msg">
  <div class="content px-4">
      @Body
  </div>
</CascadingValue>
```

Custom Components

Cascading Parameters

- CascadingParameter attribute used by a child component to access the value

```
[CascadingParameter]
public string Msg { get; set; } = string.Empty;
```


Comprehensive Blazor Server

Security and Identity

- Overview
- Blazor Server Authentication
- State Management
- Blazor Authorization
- JSON Web Tokens (JWT)

Security and Identity

Overview

- Security scenarios differ between Blazor Server and Blazor WebAssembly apps
- Blazor WebAssembly apps run on the client
  - Authorization is only used to customize the UI
  - Cannot be used to enforce authorization access rules
- Blazor Server apps run on the server
  - Authorization checks can be used to customize the UI
  - Access rules for areas of the app and components can be enforced

Security and Identity

Blazor Server Authentication

- Authentication is handled when the SignalR connection is established
  - Can be based on a cookie or some other bearer token
- An AuthenticationStateProvider obtains authentication state data from ASP.NET Core's HttpContext.User
  - Used by AuthorizeView and CascadingAuthenticationState components

Security and Identity

State Management

- A Blazor Server app maintains state for a user in server memory that is associated with a circuit
- If a user experiences a temporary network connection loss, Blazor attempts to connect the user to their original circuit with their original state
- When a user can't be reconnected to their original circuit, the user receives a new circuit with an empty state
- To preserve state across circuits, the app must persist the data to some other storage location

Security and Identity

State Management

- Different locations can be used for persisting state
  - Server-side storage
  - URL
  - Browser storage
  - In-memory state container service

Security and Identity

State Management

- For browser storage, two options exist
  - localStorage – Can persist across browser sessions
  - sessionStorage – State persists across reloads but not if the current tab is closed
- ASP.NET Core Protected Browser Storage uses ASP.NET Core Data Protection in combination with localStorage and sessionStorage

Security and Identity

Blazor Authorization

- Using an AuthorizeRouteView instead of a RouteView in App.razor will make it easy to add authorization rules to pages
  - NotAuthorized component can be used to provide custom content when the user is not authorized
- Wrapping the Router component in a CascadingAuthenticationState component makes it easy to obtain information about the authentication state at any time
  - Accessed via the context property

Security and Identity

Blazor Authorization

- Different content can be displayed based on the authentication state with the AuthorizeView component

```
<AuthorizeView>
  <Authorized>
    <p>You are authorized!</p>
  </Authorized>
  <NotAuthorized>
    <p>You are not authorized</p>
  </NotAuthorized>
</AuthorizeView>
```

```
<AuthorizeView Roles="Admin">
  ...
</AuthorizeView>
```


Security and Identity

Blazor Authorization

- An app can require authorization across the entire app by adding the [Authorize] attribute to the _Imports.razor file

```
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
```

```
@attribute [Authorize(Roles="Admin")]
```

- The authentication component should be sure to allow anonymous access

```
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@attribute [AllowAnonymous]
```

Security and Identity

Blazor Authentication

- The authentication component will often redirect the user to an authorization endpoint (identity provider) outside of the app
  - Will include a login callback to receive the response
- The IP will generate a token and redirect the user to the provided login callback
- The app will process the callback
  - Store the access token in local storage
  - Notify the ApplicationStateProvider

Security and Identity

JSON Web Tokens (JWT)

- JSON Web Token (JWT) is an open, industry standard (RFC 7519) method for representing claims security between two parties
- Claims can be verified and trusted because the token is digitally signed
  - Using a shared secret (HMAC) or with a public/private key pair
- Tokens can be encrypted but typically do not need to be

Security and Identity

JSON Web Tokens (JWT)

- The token header specifies the type of the token and the signing algorithm being used

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- The header is Base64Url encoded and forms the first part of the JWT

Security and Identity

JSON Web Tokens (JWT)

- The second part of the token is the payload which contains the claims
- There are a set of predefined claims which are not mandatory but are recommended
  - iss: Issuer
  - exp: Expiration time
  - sub: Subject
    - Unique identifier of the authenticated entity
  - aud: Audience
    - Identifies the recipients that the JWT is intended for

Security and Identity

JSON Web Tokens (JWT)

- The third part of the token is the signature
- Generated using the encoded header, encoded payload, a secret, and the algorithm specified in the header
- Ensures that the token has not been altered
- A secure pre-shared secret can provide verification of a known issuing authority
- The issuing authority's public key can be used to perform verification if signed with the authority's private key

Security and Identity

JSON Web Tokens (JWT)

- The token issuing server can be an external identity provider (Facebook, Twitter, Microsoft, etc.) or you can create your own using ASP.NET Core
- To work with JWTs, add the Microsoft.AspNetCore.Authentication.JwtBearer package
- If creating your own token issuing server, it may be a good idea to use a third-party library to handle many of the security-related details
  - IdentityServer

Security and Identity

JSON Web Tokens (JWT)

- If acting as a token issuing authority, you will need to generate the token with the appropriate claims

```
var key = new SymetricSecurityKey(
  Encoding.UTF8.GetBytes(_configuration["jwt_key"].Value);

var creds = new SigningCredentials(
  key, SecurityAlgorithms.HmacSha512Signature);

var token = new JwtSecurityToken(
  claims: claims,
  expires: DateTime.Now.AddDays(1),
  signingCredentials: creds);

return new JwtSecurityTokenHandler().WriteToken(token);
```


Security and Identity

JSON Web Tokens (JWT)

- To authenticate, you will need to configure the authentication service and middleware

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
  .AddJwtBarer(options => {
    options.TokenValidationParameters = new TokenValidationParameters
    {
      ValidateIssuerSigningKey = true,
      IssuerSigningKey = Encoding.UTF8.GetBytes(_config["jwt_key"].Value),
      ValidateIssuer = true,
      ValidateAudience = true,
      ValidateLifetime = true
    };
  });
```

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

Security and Identity

JSON Web Tokens (JWT)

- Blazor app may need to pass the access token as an HTTP header when making external API calls

```
_httpClient.DefaultRequestHeaders.Authorization =
    new AuthenticationHeaderValue("Bearer", token);
```


Comprehensive Blazor Server

JavaScript Interop

- Overview
- JavaScript Initializers
- Location of JavaScript
- Call JavaScript from .NET
- Call .NET from JavaScript

JavaScript Interop

Overview

- A Blazor app can invoke JavaScript functions from .NET code and JavaScript code can invoke .NET functions
- Use caution when mutating the DOM from JavaScript
  - Undefined behavior can occur if the state of the DOM no longer matches Blazor's internal representation
- JS interop calls are asynchronous by default to ensure compatibility across both Blazor hosting models
  - For Blazor Server, calls are sent over a network connection

JavaScript Interop

JavaScript Initializers

- It is easy to define JavaScript code that will be automatically triggered by Blazor lifecycle events
- Create a JavaScript file in the wwwroot named [assembly].lib.module.js

```
export function beforeStart(options, extensions) {
    console.log("beforeStart");
}

export function afterStarted(blazor) {
    console.log("afterStarted");
}
```

JavaScript Interop

Location of JavaScript

- External JavaScript files that will be used throughout an app should be loaded after the Blazor script reference in the host page

```
  <script src="_framework/blazor.webassembly.js"></script>
  <script src="js/site.js"></script>
</body>
```


JavaScript Interop

Location of JavaScript

- JavaScript that is only used by a specific component can be collocated with the component
- Use the filename of the component with .js appended
  - When published, the script will be moved to the web root
- Import the JavaScript file in the OnAfterRenderAsync method

```
module = await JS.InvokeAsync<IJSObjectReference>(
    "import", "./Pages/Index.razor.js");
```

JavaScript Interop

Call JavaScript from .NET

- To call JavaScript from .NET, an IJSRuntime instance is required
- Use dependency injection to obtain the IJSRuntime

```
@page "/fetchdata"
@inject IJSRuntime JS
```

- App-level JavaScript functions will be available directly via the IJSRuntime

```
await JS.InvokeVoidAsync("doSomething", someVal);
```


JavaScript Interop

Call JavaScript from .NET

- For component collocated JavaScript, use an IJSObjectReference that gets set in OnAfterRenderAsync

```
private IJSObjectReference? module;

protected override async Task OnAfterRenderAsync(bool firstRender)
{
  if (firstRender) {
    module = await JS.InvokeAsync<IJSObjectReference>("import",
      "./Pages/Index.razor.js ");
  }
}
```

```
await module.InvokeAsync<string>("doSomething", someValue);
```

JavaScript Interop

Call .NET from JavaScript

- To invoke a static .NET method from JavaScript, use DotNet.invokeMethod or DotNet.invokeMethodAsync

```
DotNet.invokeMethodAsync('{ASSEMBLY NAME}',
  '{.NET METHOD ID}', {ARGUMENTS});
```

- The .NET method must be public, static, and have the [JSInvokable] attribute

```
[JSInvokable]
public static Task<int[]> ReturnArrayAsync()
{
  return Task.FromResult(new int[] { 1, 2, 3 });
}
```


JavaScript Interop

Call .NET from JavaScript

```
<button onclick="returnArrayAsync()">
  Trigger .NET static method
</button>
```

```
<script>
  window.returnArrayAsync = () => {
    DotNet.invokeMethodAsync('MyBlazorApp', 'ReturnArrayAsync')
      .then(data => {
        console.log(data);
      });
    };
</script>
```

JavaScript Interop

Call .NET from JavaScript

- To invoke an instance .NET method, a DotNetObjectReference is used

```
@code {
  private DotNetObjectReference<FetchData>? dotNetProxy;

  public async Task TriggerDotNetInstanceMethod()
  {
    dotNetProxy = DotNetObjectReference.Create(this);
    result = await JS.InvokeAsync<string>("sayHello", dotNetProxy);
  }

  [JSInvokable]
  public string GetHelloMessage() => $"Hello, {name}!";

  public void Dispose()
  {
    dotNetProxy?.Dispose();
  }
}
```


JavaScript Interop

Call .NET from JavaScript

- To invoke an instance .NET method, a DotNetObjectReference must be used

```
<script>
  window.sayHello = (dotNetProxy) => {
    return dotNetProxy.invokeMethodAsync('GetHelloMessage');
  };
</script>
```

Comprehensive Blazor Server

Testing

- Introduction
- Unit Testing
- xUnit
- Testing Server-Side Controllers
- Testing Blazor Components
- 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

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

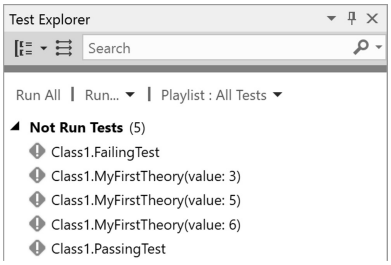
```
[Fact]
public void PassingTest()
{
  Assert.Equal(4, Add(2, 2));
}
```

```
[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 Server-Side Controllers

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

Testing

Testing Blazor Components

- To write units tests for Blazor components within an xUnit project, the bUnit package can be used

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