ASP.NET Core 3 Development

Agenda

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

ASP.NET Core 3 Development

Agenda

- Data Validation
- Error Handling
- Logging
- Testing
- Authentication
- Web APIs
- gRPC
- Blazor
- Deployment

ASP.NET Core 3 Development

Introduction

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

Introduction

What is .NET Core?

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

- `github.com/dotnet/core`

Introduction

What is .NET Core?

- .NET Core Runtime
  - Different version for each platform
  - Includes the components needed to run a console app
  - Provides assembly loading, garbage collection, JIT compilation of IL code, and other runtime services
  - Includes the dotnet tool for launching applications

Introduction

What is .NET Core?

- ASP.NET Core Runtime
  - Includes the .NET Core Runtime
  - Also includes the components necessary for running web/server applications
- Desktop Runtime
  - Includes the .NET Core Runtime
  - Also includes the components necessary for running Windows desktop applications
  - Only available for Windows

Introduction

What is .NET Core?

- .NET Core SDK
  - Includes all the available runtimes for the platform
  - Additional command-line tools including the C# and F# compilers
  - Also includes Visual Studio support

- `dotnet.microsoft.com/download`

Introduction

What is .NET Core?

- Each version of .NET Core has a lifecycle status
  - Current – Includes the latest features and bug fixes
  - LTS (Long-Term Support) – Has an extended support period
  - Maintenance – No longer current but still supported
  - Preview – Not supported for production use
  - End of Life – No longer supported

- `dotnet.microsoft.com/download/dotnet-core`

Introduction

.NET Core vs .NET Framework

- .NET Core does not support all app-models supported by .NET Framework
  - Web Forms and WCF are not available in .NET Core
  - WPF and Windows Forms are now supported but only on Windows platforms
- .NET Core contains a large subset of the .NET Framework Base Class Library
  - .NET API Browser can be used to identify the availability of specific APIs
  - `docs.microsoft.com/en-us/dotnet/api/`

Introduction

.NET Core vs .NET Framework

- It may not be possible to use .NET Core if the application depends on functionality that is not available in .NET Core
  - Including 3rd-party libraries used by the application

Introduction

Overview of ASP.NET Core

- 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

ASP.NET Core 3 Development

.NET Core SDK

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

.NET Core SDK

Installation

- The .NET Core SDK is distributed using each supported platform's native install mechanism
- Requires administrative privileges to install
- A list of installed SDK versions is available by using the .NET Core 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 Core 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": "2.1.500"
  }
}
```

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

.NET Core SDK

Version Management

- Use of global.json files can allow developers to experiment with newer versions of the SDK while ensuring consistency for specific projects
- Include a global.json file in the 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 of the system

.NET Core 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>netcoreapp3.1</TargetFramework>
</PropertyGroup>
```

.NET Core 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 3 of the SDK to build an application that targets the .NET Core 2.1 runtime

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

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

.NET Core 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 Core SDK

Command-Line Interface (CLI)

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

.NET Core SDK

Command-Line Interface (CLI)

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

Lab 1

.NET Core SDK

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

ASP.NET Core 3 Development

Application Architecture

- NuGet Packages
- Application Startup
- Hosting Environments
- Middleware and the Request Pipeline
- Services and Dependency Injection
- MVC vs. Razor Pages

Application Architecture

NuGet Packages

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

```
<PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.0" />
```


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 course 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 SDK or runtime is installed
  - More on this topic later in the section on deployment

Application Architecture

Application Startup

- When an ASP.NET Core application is launched, the first code executed is the application's Main method
  - The Main method configures and launches a host
- Previous versions of ASP.NET Core used Web Host
  - Recommendation is to now use the new .NET Core Generic Host

Application Architecture

Application Startup

- When CreateDefaultBuilder is used to create the host:
  - Sets the content root path to the path returned by GetCurrentDirectory
  - Loads host configuration from:
    - Environment variables prefixed with DOTNET_
    - Command-line arguments
  - Loads app configuration
  - Adds logging providers

Application Architecture

Application Startup

- All default host behavior can be customized by calling ConfigureHostConfiguration

```
Host.CreateDefaultBuilder(args)
  .ConfigureHostConfiguration(configHost =>
  {
    configHost.SetBasePath(Directory.GetCurrentDirectory());
    configHost.AddJsonFile("hostsettings.json", optional: true);
    configHost.AddEnvironmentVariables(prefix: "PREFIX_");
    configHost.AddCommandLine(args);
});
```

Application Architecture

Application Startup

- For an HTTP workload, CreateDefaultBuilder should call ConfigureWebHostDefaults:
  - Loads host configuration from environment variables prefixed with ASPNETCORE_
  - Sets Kestrel as the web server
  - Adds Host Filtering middleware
  - Adds Forwarded Headers middleware
  - Enables IIS integration

Application Architecture

Application Startup

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

```
webBuilder.UseStartup<Startup>();
```

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

Application Architecture

Hosting Environments

- Inject the IWebHostEnvironment service into a class to get the following information:
  - ApplicationName
  - EnvironmentName
  - ContentRootPath
  - WebRootPath

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


Application Architecture

Hosting Environments

- The 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 set using a launchSettings.json file or the command-line

Application Architecture

Middleware

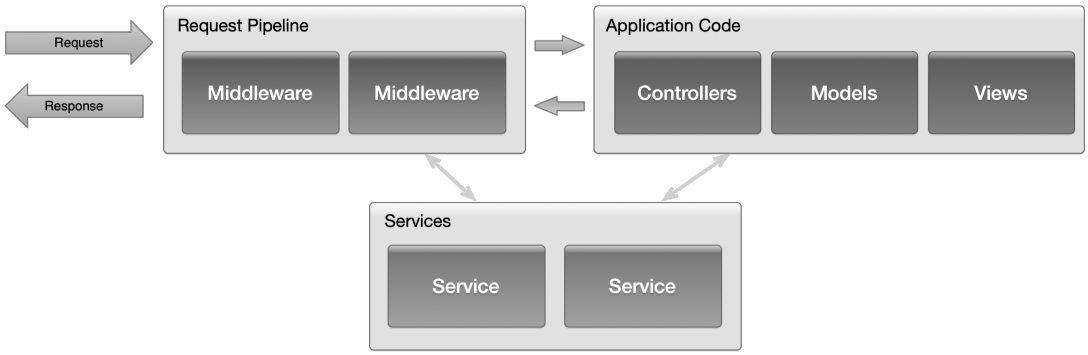
- ASP.NET uses a modular request processing pipeline
- The pipeline is composed of middleware components
- Each middleware component is responsible for invoking the next component in the pipeline or short-circuiting the chain
- Examples of middleware include…
  - Request routing
  - Handling of static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

MVC vs. Razor Pages

- ASP.NET Core 3 includes two different options when it comes to the organization of your application's server-side code
  - Model-View-Controller (MVC)
  - Razor Pages

Application Architecture

MVC

- If using MVC, your ASP.NET Core application is separated into three groups of components:
  - Models
  - Views
  - Controllers
- This helps to maintain a clear separation of concerns
- Supports TDD-friendly development

Application Architecture

Razor Pages

- Razor Pages allow for a more page-focused development style
- Closer to the organization used by WebForms in that each page has a code-behind class
- In effect, the code-behind class acts as both the controller and the model
  - Contains the first code to execute when the page is requested (like a controller)
  - Contains the data used by the view (like a model)

Application Architecture

MVC vs. Razor Pages

- Both styles can be used to build any application
- It is mostly a matter of developer preference
- The two styles can be mixed within the same application

Application Architecture

MVC vs. Razor Pages

- The lab exercises in this course use MVC but there is a partial implementation of the final lab solution in the Extras folder that uses Razor Pages
- The vast majority of the topics covered in this course are the same for MVC and Razor Pages

Lab 2

Application Architecture

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

ASP.NET Core 3 Development

Application Configuration

- Configure and ConfigureServices Method
- Configuration Providers and Sources
- Configuration API
- Options Pattern
- HTTPS and HTTP/2

Application Configuration

Configure Method

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

Application Configuration

Configure Method

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

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

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

Application Configuration

ConfigureServices Method

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

Application Configuration

ConfigureServices Method

- Services are typically added via extension methods available on IServiceCollection

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

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

Application Configuration

ConfigureServices Method

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

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


Application Configuration

ConfigureServices Method

- To enable features required by MVC, the ConfigureServices methods should include one of the following:
  - AddControllersWithViews( ) – If using views
  - AddControllers( ) – If not using views (e.g. API routing)
- For Razor Pages, use AddRazorPages( )
- MVC and Razor Pages can be used in the same application

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

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 introduces a completely new configuration infrastructure
  - Based on key-value pairs gathered by a collection of configuration providers that read from a variety of different configuration sources

Application Configuration

Configuration Providers and Sources

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

Application Configuration

Configuration Providers and Sources

- The default IWebHostBuilder adds providers to read settings from:
  - appsettings.json
  - appsettings.{Environment}.json
  - User secrets
  - System environment variables
  - Command-line arguments
- Providers can be added or removed when constructing the host

Application Configuration

Configuration API

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

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

Application Configuration

Configuration API

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

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

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


Application Configuration

Options Pattern

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

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

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

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

Application Configuration

HTTPS and HTTP/2

- It is now a general best practice to use HTTPS for your entire web application
- Also listen for HTTP but redirect (302) any HTTP request to the HTTPS equivalent
  - This redirect can be returned before reaching the web server (e.g. load-balancer or proxy) or the HttpRedirection middleware can be used

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


Application Configuration

HTTPS and HTTP/2

- If possible, HSTS should also be used in a production environment
- Adds a Strict-Transport-Security header with an expiration time
  - Once received for a domain, the browser will perform the HTTP-HTTPS redirect locally
  - Reduces the possibility of a man-in-the-middle attack that can take place during the redirection
- Header can be added by load-balancer/proxy or by using the Hsts middleware

```
app.UseHsts();
```

Application Configuration

HTTPS and HTTP/2

- The Kestrel web server supports HTTP/2 if:
  - Windows Server 2016/Windows 10 (or later) or Linux with OpenSSL 1.0.2 or later
  - Connection is using TLS 1.2 or later

ASP.NET Core 3 Development

Request Routing

- RESTful Services
- Endpoint Routing
- Route Templates
- Route Constraints
- Attribute-Based Routing

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 URIs to endpoints and dispatching incoming requests to those endpoints
- Routing can also be used to generate URLs that map to endpoints (e.g. to create a link in a view)

Request Routing

Endpoint Routing

- The EndpointRouting middleware is responsible for maintaining a list of valid routes for all subsystems used within the application (MVC, RazorPages, SignalR, gRPC, Blazor)

```
app.UseRouting();
```

- Routes are added to the route table in UseEndpoints( )

```
app.UseEndpoints(endpoints =>
{
  // adds controller actions that have a Route attribute
  endpoints.MapControllers();

  // adds Razor Pages based on the name of each page
  endpoints.MapRazorPages();
});
```


Request Routing

Endpoint Routing

- UseEndpoints can also be used to add custom routes

```
endpoints.MapControllerRoute(
  name: "privacy",
  pattern: "privacypolicy",
  defaults: new { controller = "Home", action = "Privacy" });
```

Request Routing

Endpoint Routing

- The default MVC template defines a route that can dispatch to any controller action

```
endpoints.MapControllerRoute(
  name: "default",
  pattern: "{controller=Home}/{action=Index}/{id?}");
```

- Specifies default values for controller (Home) and action (Index) that are to be used when the action (or both) are not supplied

Request Routing

Route Templates

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

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

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

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

Request Routing

Route Templates

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

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

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

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


Request Routing

Route Templates

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

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

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

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

Request Routing

Route Constraints

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

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

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

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

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


Request Routing

Route Constraints (Partial List)

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

Request Routing

Route Constraints

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

Request Routing

Attribute-Based Routing

- MVC also includes attribute-based routing
- Attribute-based routing allows you to decorate a controller action with an attribute that specifies the route for that action

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

- Added to the route table with MapControllers( )

Request Routing

Attribute-Based Routing

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

Lab 3

Application Architecture

- Refactor the application to use attribute-based routing

ASP.NET Core 3 Development

Models

- Introduction
- Persistence Ignorance
- Object-Relational Mapping
- Entity Framework Core
- Dapper ORM

Lab 4

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
- ViewModels are often used to send model data to a view
- The term Data Transfer Object (DTO) is sometimes used to represent an object that carries data between different processes or subsystems

Models

Introduction

- In some applications, it may make sense to have separate collections of entities, view models, and DTOs
- In other applications, one class may be able to act as the entity, view model, and DTO for a particular model object

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 Core, the data access component should be made available as a service via dependency injection
  - Can make it possible to test components independently with hard-coded data (no database)

Models

Persistence Ignorance

- In can also help to define entities and data access components in a separate .NET assembly (class library project)
- Makes reuse easier and helps to maintain a clear separation of concerns

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 using ADO.NET, or several frameworks exist that can help with this task
  - Entity Framework Core
  - Dapper (Micro-ORM)
  - AutoMapper (mapping one object to another)

Models

Entity Framework Core

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

Models

Entity Framework Core

- EF Core 3 contains some significant improvements compared to previous versions of EF Core
  - Rearchitected LINQ provider that generates more efficient SQL queries
  - Restricted client evaluation
  - Single SQL statement per LINQ query
  - Asynchronous streams and nullable reference types
  - New interception API

docs.microsoft.com/en-us/ef/core
/what-is-new/ef-core-3.0

Models

Entity Framework Core

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

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

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

Models

Entity Framework Core

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

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


Models

Entity Framework Core

- To specify different mappings, you can use data annotations on your 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

- By default, EF Core will write 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 entities can be defined manually, or the the code can be automatically generated by using the EF Core…
  - CLI tools
  - Package Manager Console tools in Visual Studio

Models

Entity Framework Core

- To achieve complete separation between the data access code and the web application, it can be helpful to define a separate interface for the data access component
- Provides more flexibility when it comes to…
  - Unit testing
  - Comparing different data access technologies
  - Switching to a different persistence mechanism (e.g. microservices)
- The interface should be defined in a separate class library so that different data access libraries and applications can all reference it

Models

Entity Framework Core

- Achieving this degree of separation does require more effort
  - Additional class libraries and more API surface area to maintain
  - Difficult to use a feature that is not supported across all data access implementations (e.g. change-tracking)
- This decision should be made on a case-by-case basis
- We will use this approach in the labs so that we can easily demonstrate other options for the data access layer

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/`

Lab 5

Data Access

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

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 but there is a version of the lab project in the Extras folder that uses Dapper instead of EF
  - There is also a version that uses raw ADO.NET

ASP.NET Core 3 Development

Controllers

- Responsibilities
- Requirements and Conventions
- Dependencies
- Action Results

Controllers

Responsibilities

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

Controllers

Responsibilities

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

Controllers

Requirements and Conventions

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

Controllers

Dependencies

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

```
public class HomeController
{
  private IEmailSender _emailSender;

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


Controllers

Action Results

- Although not required, controller actions typically return an instance that implements IActionResult

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

- Framework uses the action result to create an HTTP response

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 through the use of ToString

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

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

Controllers

Action Results

- A controller action may invoke an asynchronous method to perform an IO-bound operation
  - Database access, web service call, etc.
- The action can be marked as async with a return type of Task<IActionResult>
  - Allows for the calling of other asynchronous methods using await

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


Controllers

Action Results

- Making an action asynchronous does not change the experience for the user
  - No response is sent to the client 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

Lab 6

Controllers

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

ASP.NET Core 3 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

Views

Conventions

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

```
return View();
```

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

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

Views

Conventions

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

Views

Conventions

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

Views

Razor Syntax

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

Views

Razor Syntax

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

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

Views

Razor Syntax

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

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


Views

Razor Syntax

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

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

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

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

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

Views

Razor Syntax

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

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

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


Views

Razor Syntax

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

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

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

Views

Razor Syntax

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

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

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

Views

Layouts

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

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

Views

Layouts

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

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

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

Views

Layouts

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

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

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

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

Views

View Data

- 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.DataAccess.Product>
```

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

```
@using EComm.DataAccess
```


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 (Part I)

- Modify the home page to display a list of products

ASP.NET Core 3 Development

Views (Part II)

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

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


Views

Tag Helpers

- ASP.NET Core 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 db.Products.ToListAsync();
  return View(products);
}
```

Views

View Components

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

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

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

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

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


Views

View Components

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

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

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

Views

View Components

- View components can exist in any project folder / 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 3rd-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
  - To enable in a Visual Studio project, right-click on the project and select [Manage Client-Side Libraries…]
  - To add a new client-side library, right-click on the project and select [Add > Client-Side Library…]

Views

Client-Side Dependencies

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

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


Views

Client-Side Dependencies

- ASP.NET Core views can conditionality 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

Razor Pages

- A partial implementation using Razor Pages of the final lab project is in the Extras folder

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 can also result in much more work in some cases

Views

View Models

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

ASP.NET Core 3 Development

HTML Forms

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

HTML Forms

Introduction

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

HTML Forms

Form Tag Helper

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

```
<form asp-controller="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 = {new value};
product.UnitPrice = {new value};
// set other properties
await SaveChangesAsync();
```

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

Lab 11

HTML Forms

- Complete the ability to edit a product

ASP.NET Core 3 Development

Data Validation

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

Data Validation

Introduction

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

Data Validation

Data Annotations

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

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

Data Validation

Data Annotations

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

Data Validation

Input Tag Helpers

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

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

- 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

Data Validation

Model Binding

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

Data Validation

Validation Tag Helpers

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

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

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

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

Data Validation

IValidatableObject

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

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


Lab 12

Data Validation

- Add data validation for when editing a product

ASP.NET Core 3 Development

Application State

- Introduction
- HttpContext.Items
- Session State
- TempData

Application State

Introduction

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

Application State

Introduction

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

Application State

HttpContext.Items

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

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

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

Application State

Session State

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

Application State

Session State

- Session must be configured in your Startup class

```
services.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

Lab 13

Application State

- Implement shopping cart functionality (Part 1)

Lab 14

Application State

- Implement shopping cart functionality (Part 2)

Lab 15

Application State

- Implement shopping cart functionality (Part 3)

ASP.NET Core 3 Development

Error Handling

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

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

- Best practice is for exception pages to consist of only static content
- This minimizes the possibility of the exception page itself throwing an exception

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

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

ASP.NET Core 3 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 3rd-party logging frameworks can be used to provide 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

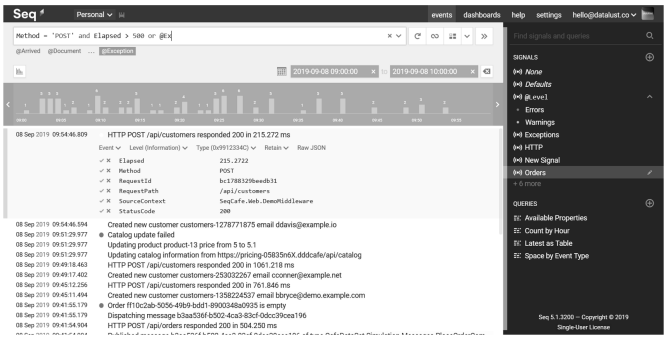
- Serilog has become a popular choice for ASP.NET Core
  - Wide variety of destinations and formats
  - Can record structured event data

github.com/serilog/serilog-aspnetcore

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



Logging

Conclusion

- One important note is that the logging framework(s) that you choose should not change how you actually write to the log (ILogger)
  - Only the code you need to include in Startup.cs (and maybe Program.cs)

ASP.NET Core 3 Development

Testing

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

Testing

Introduction

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

Testing

Introduction

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

Testing

Unit Testing

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

Testing

Unit Testing

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

Testing

Unit Testing

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

Testing

Unit Testing

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

Testing

Unit Testing

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

Testing

xUnit

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

Testing

xUnit

- Fact attribute is used to define a test 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

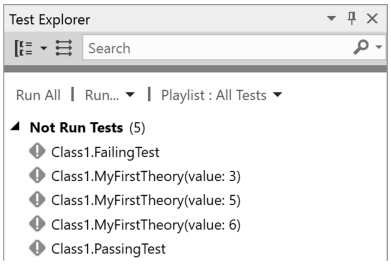
```
[Fact]
public void PassingTest()
{
  Assert.Equal(4, Add(2, 2));
}
```

```
[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

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

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

```
public class ECommAppFactory<TStartup> :
  : WebApplicationFactory<TStatup>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 3 Development

Authentication

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

Authentication

Introduction

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

Authentication

ASP.NET Core Identity

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

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

Authentication

Cookie Middleware

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

Authentication

Cookie Middleware

- 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();
```

Authentication

Cookie Middleware

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

Authentication

Cookie Middleware

- 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

Authentication

Cookie Middleware

- Once authenticated, you 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));
```


Authentication

Cookie Middleware

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

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

- The SignOutAsync method is also available

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

Authentication

Authorization

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

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

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


Authentication

Claims-Based Authorization

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

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

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

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

Authentication

Claims-Based Authorization

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

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

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


Lab 18

Authentication

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

ASP.NET Core 3 Development

Web APIs

- API Controllers
- Testing APIs
- Retrieval Operations
- Create Operations
- Update Operations
- Delete Operations
- OpenAPI (Swagger)
- Cross-Origin Request Sharing (CORS)

Web APIs

Introduction

- ASP.NET Core provides extensive support for building Web APIs
- The separate Microsoft "Web API" framework has been merged into ASP.NET Core
- You can include a Web UI and API in the same project or use completely separate projects

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 deal with 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
  - Multipart/form-data request interface

Web APIs

API Controllers

- ASP.NET Core provides a collection of attributes that can be used when defining Web API actions

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


Web APIs

Testing APIs

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

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://localhot:5001/~ ls

Products   [get|post]
Customers  [get|post]
```

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

Web APIs

Retrieval Operations

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

```
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 19

Web API

- Build a Web API 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 use an HTTP 204 (no content) status code

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

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

Web APIs

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 20

Web API

- Build a Web API for editing a product

Web APIs

OpenAPI (Swagger)

- OpenAPI (previously known as Swagger) is a specification for describing REST APIs
  - API discoverability
  - Interactive documentation
  - Client code generation
- Swagger UI provides a web-based UI that provides information about a service
  - Generated from the API's Swagger document

Web APIs

OpenAPI (Swagger)

- Swashbuckle.AspNetCore is an open course project for generating Swagger documents
- NSwag is another open source project that can generate Swagger documents
  - Also offers the ability to generate C# and TypeScript client code for an API

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 3 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 of this course but we will add a working server and client to our project

`https://grpc.io/`

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 feature that are not yet supported
- Not currently supported on IIS or Azure App Service
  - Http.sys does not yet support trailing headers
- 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

Lab 21

gRPC

- Build a gRPC server and client for getting product information

ASP.NET Core 3 Development

Blazor

- Razor Components
- Blazor Server
- Blazor WebAssembly

Blazor

Introduction

- Blazor is a framework for building interactive client-side web UI with .NET code
- Does not require JavaScript (other than that provided by the framework)
- Provides an alternative to frameworks such as Angular and React

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 as view 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 one of two different configurations
  - Blazor Server – Supported in ASP.NET Core 3.0
  - Blazor WebAssembly – In preview for ASP.NET Core 3.1

Blazor

Blazor Server

- With Blazor Server, a JavaScript file (blazor.server.js) is sent to the browser (209 KB)
  - 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 various 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

```
devblogs.microsoft.com/aspnet/blazor-server-in-
    net-core-3-0-scenarios-and-performance/
```


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 (webassembly.org)
  - This 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 user to install something ahead of time (not a 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…
  - The WebAssembly for the runtime is automatically downloaded by the browser
    - Approximately 2 MB in size but Microsoft hopes to have the size down to less than 1 MB before release
  - The .NET assemblies for the app (containing Razor components) are downloaded (typically very small)
- No state is maintained on the server
- .NET classes like HttpClient can be used to communicate with the server

ASP.NET Core 3 Development

Deployment

- dotnet publish
- Kestrel
- IIS
- Docker

Deployment

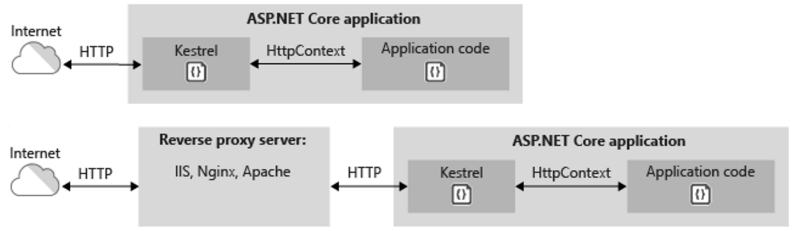
dotnet publish

- The dotnet publish command compiles an app and copies the files required into a publish folder
  - Used by Visual Studio's [Build > Publish] wizard
- When preforming 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

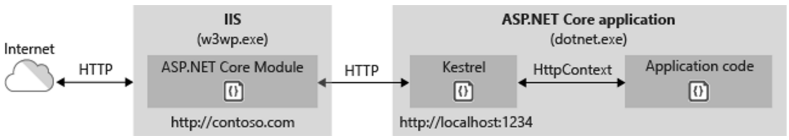
- The Kestrel web server is supported on all platforms where .NET Core is supported
- You can use Kestrel by itself or with a reverse proxy server



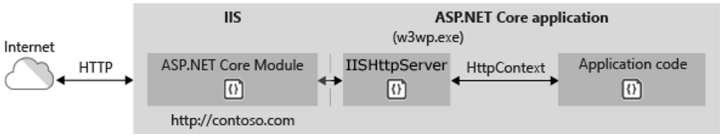
Deployment

IIS

- Out-of-process hosting



- In-process hosting



Deployment

Documentation

- There is extensive documentation for a wide variety of deployment scenarios

docs.microsoft.com/en-us/aspnet/core/host-and-deploy

Deployment

Docker

- There is extensive documentation for a wide variety of deployment scenarios

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

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 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
  - microsoft/dotnet
  - microsoft/aspnetcore

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


Deployment

Docker

- Visual Studio 2019 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 container 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