.NET Enterprise Applications

Preface:

Much of what is covered in this document has already been covered in my Java Enterprise Applications document. That being said, I encourage you to continue reading this document if it is your goal to better understand enterprise applications and MVC. Many concepts will be repeated from previous documents. Rather than deter you, I hope that seeing the same principles applied under a different language and framework help grow your insight.

Intro:

The .NET ecosystem (at least to me), had always come off as monolithic, daunting, and niche. Whilst not all of these are false statements, I grew to enjoy .NET more than anticipated, and I hope to share some of that with you here. If you are unaware, .NET is the name of an software/application ecosystem created by Microsoft. The term “software ecosystem” may likely be a foreign concept to you if you have not used something like .NET before. Essentially, it is comprised of various languages, frameworks, libraries, and tools that can work together or be used independently to create various sorts of applications. .NET can be used to create web apps, mobile apps, games, etc. The core languages to .NET are C++, C# (which is what we’ll be covering), F#, and VB Script (a modern take on Visual Basic). In reality, there are many more than just these, but the ones I’ve listed are the most commonly used. This document, as previously mentioned, will focus solely on creating web enterprise applications.

History:

.NET was initially released in 2002 in hopes of aggregating all of the common paradigms of Windows programming. .NET was eventually made free (as in monitary value, not as in libre) and open source. This new open source version of .NET is technically called .NET Core, but I will be referring to .NET Core plainly as .NET for simplicity. The benefit of .NET is that applications can run on a variety of architectures and platforms. This was purposefully done to compete with Java, which boasted a very similar idea. In fact, as you will soon see, Java and C# are extremely similar in terms of syntax and other principles. Since .NET became open source, both Linux and Mac now support .NET (in addition to Windows of course). .NET is updated regularly for security and quality on the second Tuesday of each month. Additionally, the Red Hat Enterprise Linux (RHEL) organization collaborates with Microsoft to ensure that .NET Core works well on Linux.

.NET SDK:

As is the case with any virtualized development tools, we require an SDK to run .NET. The SDK is simply known as the .NET SDK. This includes things like the .NET CLI (command-line tools used for local development and continuous integration scripts), the .NET driver (used for running framework-dependent apps i.e. apps that depend on .NET), the Roslyn and F# compilers, the MSBuild build engine, the .NET runtime, runtime libraries, the ASP.NET Core runtime (we will cover ASP.NET shortly), and the desktop runtime.

You’re reading my notes so I’m forcing my Linux evangelism unto you. .NET was initially created for Windows and is documented under the Microsoft Documentation (MSDN). For this reason, Windows users should have no troubles installing and configuring .NET. Linux users may be left in the dark in terms of how to install .NET. Fear not, I shall be your guide (at least, assuming you are running some sort of Arch-based distro, which probably isn’t the case, statistically speaking). Anyways, in order to get started, we first install the .NET runtime and .NET SDK. On Arch, this is done by running sudo pacman -S dotnet-runtime dotnet-sdk. One or more hidden directories will be created under your home directory. These will be named .dotnet, .aspnet, and .nuget (.aspnet and .nuget may not appear until you’ve implemented a project which uses those features). You must include the .dotnet directory in your PATH (how you do this will depend on your setup). Finally, if you are sane, you will dissable telemetry (which is turned on by default) by adding export DOTNET\_CLI\_TELEMETRY\_OPTOUT=1 to your bashrc or whichever file you wish to place it in that gets sourced automatically by your shell. Now you can use the .NET CLI tool to create projects, which we will be using going forwards.

MSBuild:

The Microsoft build engine, aptly titled MSBuild, is the build tool used for .NET applications. It provides an XML schema for your projects that controls the build settings for your project. The extension for this schema will either be .csproj, .fsproj, or .vbproj, depending on whether you’re using C#, F#, or VBScript, respectively.

NuGet:

NuGet is essentially the package mananger for .NET, not unlike pip for Python or npm for Node JS. A NuGet package is just a .zip file with the .nupkg extension that contains DLLs, files related to the package, and a manifest file containing general package information.

Common Language Infrastructure (CLI):

Common Language Infrastructure (CLI), not to be confused with Command-line Interface (also CLI), is basically the compilation pipeline for .NET. Something very interesting about .NET is that all languages which are supported by CLI can be compiled into a language called Common Intermediate Language (CIL, also not to be confused with CLI). CIL is essentially like Bytecode in Java, but for .NET. Thus the pipeline goes something as follows: CLI compliant language gets compiled to CIL, CIL gets compiled to machine code, machine code is executed by the processor. Sometimes CIL is referred to as Microsoft Intermediate Language (MSIL) or just Intermediate Language (IL) to avoid confusion with CLI.

Common Language Runtime (CLR), Just in Time (JIT), and Ahead of Time (AOT):

The .NET Common Language Runtime (CLR) is designed to run CIL code (think of the JVM). The CLR handles memory allocation and memory management (e.g. garbage collection) and is a virtual machine which generates and compiles code using a Just-in-Time compiler (JIT).

Just-in-Time and Ahead-of-Time are both components of CLR and are both compilers, but have slight differences. JIT compiles CIL to machine code, does optimizations that target a specific processor architecture/ISA, performs tiered compilation, and occurs *during* execution of the application. AOT, on the other hand, compiles the program to machine code ahead of time, meaning that it has a slight upper hand in terms of runtime speed. It can also be combined with JIT.

.NET Standard:

The .NET Standard are a set of APIs that are implemented by the Base Class Library (BCL) of .NET. You can know when you’re using a BCL if it begins with System.\*, or sometimes Microsoft.\* (though not always in that case). These libraries are, as you may guess, foundational libraries that many other things in .NET are built upon (e.g. ASP.NET). Once again, this sort of parrots the Java Class Library (JCL)... Can you tell that .NET is similar to Java yet?

Mono and Universal Windows Platform (UWP):

Mono is a different implementation of the CLR and was designed to be more lightweight. It powers Xamarin applications on Android, macOS, iOS, tvOS, and Unity. It implements the .NET Standard and was the thing that was used to run .NET on Linux prior to .NET Core.

UWP came with Windows 10. It was meant to run on all Windows 10 platforms. UWP required user authorization to run because it could execute some pretty mallicious stuff if you were not careful. The benefit was greater language support, elimination of machine rot, and optimized APIs for specific device features.

Other .NET Features:

Windows Presentation Foundation (WPF) is a hardware rendered UI layer that uses DirectX as it’s backend. Most modern windowed UI applications are developed using WPF. It is implemented using an XML-based language called XAML, which you’ve likely heard of (at least in name) before.

Windows Communication Foundation (WCF) is a communication layer which allows applications and services to communicate with each other using XML, JSON, etc. It provides bi-directional communication.

ADO.NET is a data description language that allows .NET to communicate with data sources like SQL Server, MySQL, Access, Oracle, etc.

ASP.NET will be the focus of this document, but let us cover it in brief first. Originally, we had Active Server Pages (ASP), which were used to generate dynamic content on the fly and serve it to the user. It allowed the user to interact with the web server, save session state (stateful), move data between pages, and process I/O requests. Note that ASP.NET is an old technology that was revamped with ASP.NET MVC (much like how VB was revamped into VBScript). ASP.NET Web Forms, at their peak, accounted for 30% of all web traffic. They were HTML5 compliant, and CSS3+ compliant, AJAX capable, etc. You shouldn’t worry about ASP.NET Web Forms much since they have largely been phased out in favor of the next topic, ASP.NET MVC.

ASP.NET MVC:

ASP.NET MVC can be considered as a framework for ASP.NET which implements the MVC design pattern into our web applications. It provides basic features such as session state, server rendered HTML, .NET backend, ADO.NET/Entity framework used for DB connections, etc. The benefits of this framework are that the programmer now has absolute control over the rendered HTML. It provides an implementation of the MVC design pattern that is highly scalable and more modular.

ASP.NET Razor Pages:

Finally, let us discuss ASP.NET Razor Pages. These are loosely based of the Model View ViewModel (or MVVM) design pattern. Razor is a server-side scripting language, similar to PHP, which is embedded within HTML code using special tags. These tags are read by ASP.NET and are rendered on the server. Do not worry about learning a new language because as you will soon see, Razor is practically identical to C#, and C# is practically identical to Java, syntactically speaking.

Syntactic Differences Between C# and Java:

Rather than cover the entirety of C#’s syntax (which would be quite redundant), let us cover some of the differences between the two. It should be noted that C# borrows one or two things from C++, so it helps if you’re familiar with C++ as well.

First, let’s cover the language paradigms. C# is type-safe, object-oriented, and both imperative and declarative. The reason we say that it is both imperative and declarative is because it implements something known as LINQ which are SQL-like statements that are declarative. For the most part though, C# is imperative.

I would say that the largest difference between C# and Java is that Java uses camel case notation everywhere, whereas C# uses a mix of camelCase and PascalCase. Here is a list of which tokens are traditionally written in camelCase, and which are written in PascalCase:  
  
**Class:** PascalCase

**Record:** PascalCase

**Struct:** PascalCase

**Method:** PascalCase

**Parameters:** camelCase

**Member Variables:** camelCase

When in doubt, the general philosophy behind the casing conventions of C# are to use PascalCase for public, protected, or protected internal tokens (basically anything that is meant to be accessed externally) and camelCase for private or internal tokens (anything that is not meant to be accessed outside of the class or struct). Microsoft also recommends that private or internal fields/member variables be prefixed with an underscore. If they are static, they should be prefixed with s\_, and if they are thread static, they should be prefixed with t\_. Here is an example:

public class DataService

{

private static IworkerQueue s\_workerQueue;

[ThreadStatic]

private static TimeSpan t\_timeSpan;

}

See this document for more information: <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions>

Namespaces:

In similar fashion to C++, C# uses namespaces. We declare a namespace using the namespace keyword e.g. namespace ExampleNamespace { ... }. Namespaces are heirarchical. In the same manner that all classes extend to a common root object, namespaces also extend to a default namespace referred to as the global namespace. If your program does not create a namespace, then it is assumed to exist within the global namespace. Also, in similar fashion to C++, we can use the ‘using’ directive to specify one or more namespaces that will be implicitely referenced within some scope. We can create aliases with the using directive as well e.g.: using Project = PC.MyCompany.Project;

Acceptable Method Signatures for Main():

C# provides 4 signatures for Main():

- static void Main()

- static int Main()

- static void Main(string[] args)

- static int Main(string[] args)

Print Statements and User Input:

Print statements are performed using the Write() and WriteLine() functions in the Console namspace. Therefore, to print a statement, we do Console.WriteLine(“HelloWorld”);. Similar to other languages, we can do variable expansion within our strings by prepending a dollar sign symbol before the string literal and enclosing variables within curly braces. For instance: Console.WriteLine($”My name is {name}”); will print “My name is ” followed by whatever value is stored in the variable name.

In order to read input we use Console.ReadLine() which will return a string no matter the content of the input. This brings us to types and type casting in C#.

Types:

C# is a bit unique when it comes to its type system, called the C# Common Type System (CTS). C# does not have primitives, unlike a language like C. Instead, we have *value types* and *reference types*. As you may be able to guess, reference types are objects that are allocated in heap space. Reference types include objects, strings, arrays, etc. Value types in C# are the closest thing to primitives in C#. They are allocated in stack space, but the way that this is achieved is by wrapping them in structs. Value types include: bool, byte, sbyte, char, decimal, double, float, int, uint, nint, nuint, long, ulong, short, and ushort. The ‘s’ prefix stands for signed, ‘u’ stands for unsigned, and ‘n’ stands for native-sized (based on architecture). Structs in C# function more like C++’s implementation rather than C. What I mean is that structs may contain functions or attributes that can be accessed in similar fashion to an object.

The root object for reference types is called System.Object, where System is a namespace. This mimics Java’s root Object class. Thus, some classes that inherit properties of System.Object include System.String, System.Array, System.Enum, and most interesting – System.ValueType. System.ValueType is the parent class that all value types are derrived from. For example, System.Int32, System.Boolean, etc.

Value types cannot be null because unlike reference types, they are not pointers. However, I mentioned that they are encapsulated within structs, so it is actually possible to make a value type nullable. Syntactically, this is accomplished by appending a question mark after the value type e.g. int? I = null; The question mark does not denote that the value type *must* be null, only that it is capable of being assigned the value null. Note that the syntax for marking value types as nullable is the same as the syntax for making reference types nullable.Value types are only *technically* nullable. They do not actually contain the value null, but rather, set a flag within their struct that marks them as null, tricking the system.

Value types each have a default value. For instance, integer types will default to 0. This means that we can safely use a variable even if we declare it, but don’t initialize it. Reference types, on the other hand, are pointers that contain an address in heap space that contains the data. The default value for reference types is null, and unlike value types, this is actually null i.e. address 0x00000000 (not just a flag that tells the compiler it’s null). Reference types also support inheritance, unlike value types.

Here is a list of data types and their respective sizes. Note that C# supports unicode, thus characters are always 2 bytes. There is no “official” type for ascii characters that only need to take up one byte.

int: 4 bytes

long: 8 bytes

float: 4 bytes

double: 8 bytes

bool: 1 bit

char: 2 bytes

string: 2 bytes per character

Implicit Types / Anonymous Types:

Another C++ism is the ability to create variables with implicit typing. Instead of the “auto” keyword, C++ uses “var”. C# is a completely type safe language, so implicit types cannot be broken after initialization. Additionally, as is always the case with type safe languages that support implicit types, variables declared with var *must* be initialized at the time of declaration so that the compiler can assign the variable its type based on the value that was assigned.

Anonymous types allow us to encapsulate read-only properties into a single obect without having to explicitly define a type first. For example:

var v = new { Amount = 108, Message = “Hello” };

Anonymous types cannot accept null. Their values can be accessed using the dot operator as per the usual, but since these are read-only properties, you cannot reassign them.

Type Casting:

Implicit casts in C# are handled automatically, but only work when the type that they are being cast to is larger than the current type. Thus, the flow of implicit casts looks something like: char -> int -> long -> float -> double. In order to cast to a narrower type, you must use explicit casts. Explicit casts are not guaranteed to work, unlike implicit casts. Additionally, information is likely to be lost. You may either use the common casting method of using round brackets with the desired type before the value e.g. (int)6.7 or you may use the cast functions which are provided in the Convert class e.g. char a = Convert.ToChar(97); Note that Convert is provided under the System namspace.

Classes:

Classes are really nothing new in C#. They function pretty much the same as C++. You provide a scope identifier, followed by the class keyword, class name, and then optionally a colon followed by a class or interface that you would like to inherit/implement. Unlike C++, C# does not allow multiple inheritance. Therefore, the first class that is provided after the colon is inherited, and then subsequent comma-separated interfaces may follow. C# can tell whether or not the first argument is a class or an interface, so the method of implementing a class or inheriting an interface is the same. Here is an example to illustrate:

class MyClass : Foo, IFoo, IBar

{

}

interface IFoo

{

}

interface IBar

{

}

class Foo

{

}

This is acceptable because Foo is a class and is the first argument in the comma-separated list. If it had been placed second, a compile time error would have occurred since C# would be trying to inherit from a class.

Also, note that the standard in C# is to prefix interfacing with a capital ‘I’. This provides additional clarity when determining what is a class and what is an interface.

Structs:

Structs operate in very similar fashion to classes, with the primary difference being that they cannot be derrived from other structs or classes (however, they can implement other structs or interfaces). Member variables are implicitely public by default and a public default constructor is also present if not explicitely defined. Here is an example of a struct in C#:

public struct Point

{

public double X { get; }

public double Y { get; }

public Point(double x, double y) => (X, Y) = (x, y);

}

Enums:

Enums are a bit interesting in C# since we can specify an underlying type for the enum class, however, the underlying type must be an integer-based type (this includes byte, sbyte, short, ushort, int, uint, long, and ulong). Here is a basic example of an enum:

using System;

namespace Program

{

public enum Color: byte

{

Red = 1,

Green,

Blue,

White = Red | Green | Blue

}

public class Foo

{

public static void Main()

{

Color c = Color.Red;

Console.WriteLine(c); // Outputs Red

Console.WriteLine((byte)c); // Outputs 1

}

}

}

Essentially the keyword enum replaces the keyword class. I don’t mean to say that enums and classes are identical, I just mean that they function similarly. This explains why we do not need a semi-colon to end the scope, and how we can seemingly create something similar to instances of the enum. The biggest takeaway here though, should be that C# assumes that when you access an enums fields, it ought to return a string representation of the field rather than returning its assigned value as is the case in most other languages. In order to receive the value of the field, I have to cast it to the appropriate type, as demonstrated in the second print statement.

Arrays:

Arrays are quite straightforward and should be familiar to you. Here are some examples of valid declarations:

- int[] primes;

- int[] primes = new int[9];

- int[] prime = new int[] { 1, 2, 3, 5, 7, 11, 13, 17, 19 };

- int[] prime = { 1, 2, 3, 5, 7, 11, 13, 17, 19 };

The new Keyword:

The new keyword in C# is somewhat redundant. Essentially, it invokes the appropriate constructor for the object or struct that it prepends. In the case of structs, it does not cause them to become heap allocated, it just invokes the constructor. And since C# has automatic garbage collection, a keyword like delete does not exist for objects allocated with new. The primary requirement for the new keyword is for abstract types, which we discussed earlier.

Variables, Constants, Access Modifiers:

Variable names use PascalCase as the convention. They must begin with a letter or underscore and cannot contain special characters. Microsoft decided to allow users to override keywords as variable names for some reason (personally, I find that to be a really dumb idea). To do this, just prefix the variable name with the ‘@’ symbol e.g. int @foreach;

Constants in C# are declared with the const keyword. Constants typically still use PascalCase – same as regular variables. They must be initialized at the time of declaration as per the usual.

C# has multiple access modifiers which I will list below:

**public:** Access is not limited.

**private:** Access is limited to the class.

**protected:** Access is limited to the class or classes derived from the class.

**internal:** Access is limited to the current assembly (.exe or .dll).

**protected internal:** Access is limited to the current assembly or types derived from the containing class.

**private protected:** Access is limited to the containing class or types derived from the containing class withing the current assembly.

Foreach Statement:

I am highlighting the foreach statement in C# because it is one of the few things that deviates slightly from Java’s syntax. All that you need to know primarily is that the colon in Java’s foreach loop gets replaced with the keyword “in”. So rather than foreach (int i : list) { ... } it would be foreach (int i in list) { ... }. The way that this is handled under the hood is that C# creates an IEnumerable which enumerates through each item in the collection.

Exception Handling:

Same as Java, exceptions follow the sequence try, catch, finally. To refresh your memory, the try block attempts to run some code, the catch block runs if an exception is caught in the try block, and the finally block runs after the first two blocks regardless of whether or not an exception was caught. Exceptions derive from the System.Exception class. The throw keyword allows us to manually throw an exception e.g. throw new Exception();

Properties and Indexers:

One thing that I personally really like about C# is that it allows us to create properties and indexers within a class. The way that this is implemented might look something like the following:

class Person

{

private string name; // field

public string Name // property

{

get { return name; } // get method

set { name = value; } // set method

}

}

If you’re attentitive, you’ll notice that the member variable “name” uses lower case letters. You may also notice that it is private. This is because name (the field) is not actually intended to be accessed externally, as is usually the case in OOP languages. It is simply there to store data. The property for the name field is capitalized so that it looks as though we are accessing the field when, in fact, we are not. Two pseudo-functions are created which perform get and set operations respectively via the property “Name”. Note that this code can be condensed into the following:

class Person

{

public string Name { get; set; }

}

This is because C# knows how to automatically fill in getters and setters for us. The member variable “name” is implicitely created by the compiler.

Indexers are a bit more niche. They allow us to create a class, struct, or interface that client applications can access as an array. The compiler generates an Item property with getters and setters for us. Indexers are mostly used in types whos primary purpose is to encapsulate an internal collection. For example, assume a class called TempRecord is created and contains a float array called temps as one of its fields which stores temperatures. Now assume we create an intance of TempRecord called tempRecord. If an indexer is not present in TempRecord, then to access the 5th element of temps, we must do something like: float temp = tempRecord.temps[4]; However, with an indexer, we can do: float temp = tempRecord[4];

Declaring an indexer looks something like the following:

public int this[int index]

{

get => temps[index];

set => temps[index] = value;

}

Notice how this is similar to a regular property, but this time we use the keyword “this” rather than the field name and we provide it with the [int index] clause to suggest that it can accept a parameter which will be our index. As I briefly mentioned, this will create a property called Item on the object which is not directly accessible from the class instance.

Language Integrated Query (LINQ):

Here is where we come to one of the useful features exclusive to C#. Language Integrated Queries (LINQ) are declarative statements which mimic SQL queries. They work on generic collections, databases, and XML documents. There is also something known as parallel LINQ which gives us the ability to run certain parts of a query in parallel, resulting in faster performance in some cases. The query syntax follows the following pattern:

from <range variable> in <IEnumerable<T> or IQueryable<T> collection>

<standard query operators><lamda expression>

<select or groupBy operator><result formation>

For example:

(from w in Words where w.Contains(“e”) && w.Length > 4 select w.Length).ToList();

This would take all of the items in the Words collection and check if they a) contain the letter e, and b) are greater than 4 letters. The select w.Length operation returns the length of the word if the prior conditions were true. Finally, since LINQ queries return an IEnumerable<T> object, we need to always cast to the type IList<T> using the ToList() method. Here is a full list of query operations you can perform using LINQ: <https://www.tutorialsteacher.com/linq/linq-standard-query-operators>.  
  
 There are two methods of writing LINQ queries. The first method is the standard method that I gave in the example above. The second method is to use lambda functions. It will be beneficial if I show you a comparison between some queries in both their normal and lamda format.

**Normal Format:**

1. var simpleSelectEnumerable = (from w in Words select w);

2. var simpleSelect = (from w in Words select w).ToList();

3. var selectWithLogic = (from w in Words where w.Contains(“e”) select w).ToList();

4. var selectWithCompoundedLogic = (from w in Words where w.Contains(‘e’) && w.Length > 4 select w).ToList();

5. var selectWithLogicAndOrderBy = (from w in Words where w.Contains(‘e’) orderby w select w).ToList();

6. var selectPropertiesOfTheObjectsBeingQueried = (from w in Words where w.Contains(‘e’) && w.Length > 4 select w.Length).ToList();

7. var max = (from w in Words select w.Length).Max();

8. var min = (from w in Words select w.Length).Min();

9. var sum = (from w in Words select w.Length).Sum();

10. var average = (from w in Words select w.Length).Average();

11. var take = (from w in Words select w.Length).Take(2).ToList();

**Lambda Format:**

1. var simpleSelect = Words.ToList();

3. var selectWithLogic = Words.Where(w => w.Contains(‘e’)).ToList();

4. var selectWithCompoundedLogic = Words.Where(w => w.Contains(‘e’)).OrderBy(w => w).ToList();

6. var selectPropertiesOfTheObjectsBeingQueried = Words.Where(w =>

{

return w.Contains(“e”) && w.Length > 4;

}).Select(w => w.Length).ToList();

7. var max = Words.Max();

8. var min = Words.Min();

9. var sum = Words.Select(w => w.Length).Sum();

10. var average = Words.Select(w => w.Length).Average();

11. var take = Words.Take(2).ToList();

Back to Basics – What is a Website?:

We’ll start at the beginning, by asking ourselves what a website is. Well, a website in the most basic form is one or more HTML files that get parsed by the browser to display content. During web 1.0 the web was comprised of static sites that had limited interaction with the user. Typical static sites include blogs, recipes, how-to’s, etc.

HTML v3 introduced the formalized concept of collection data from users via forms. This was accomplished through something called the Common Gateway Interface (CGI). At this time, scripting languages began to evolve (EMCA Script a.k.a. JavaScript, PHP, and ASP).

What is a Web Application?:

Websites started to become more complex as time went on. Languages like ASP and PHP allowed programmers to interact with the server. Dynamic “server-side” execution allowed for advanced techniques like querying a database and returning the result to the user. This was the begginning of web 2.0. Nowadays, we have pre-compiled executables installed as applications within webservers that use HTTP (among other technologies) as the mechanism to communicate between the browser (client) and the application (server).

Active Server Pages (ASP) and Personal Homepage (PHP):

The first incarnations of “data driven/server-side” web programming were ASP and PHP. Both of these are server-side scripting languages used to interact with their respective web servers (Apache and IIS). They provide the ability to render HTML content on the server and send it to the client. They also provide RESTful API services to perform CRUD operations on the server from the client. Finally, they allow programmers to interact with data sources such as databases, XML, etc.

Review – MVC:

MVC stands for Model, View, Controller. There is also MVVM (Model, View, ViewModel) which we will look into later. MVC is a design pattern used to split logic into distinct categories. The Controller essentially handles I/O from the client and is a mediator between the View and the Model. In ASP.NET MVC (a framework for ASP.NET that implements the MVC design pattern,) the functions which are contained within the Controller are called actions. User requests are routed to a Controller which then chooses which View to render and provides it with any Model data that it requires. The Model then, represents the state of the application. We will look into the .NET Entity Framework later on, but to summarize, the Model is where our database tables will get mapped to. Typically a model contains a class to represent the state and contains the methods/business logic to be performed by the application. The Model can be tailored to send data towards a specific view in the form of a ViewModel, which is essentially just a lightweight class containing our data. Finally, the View is responsible for presenting content through the user interface. In ASP.NET, Views rely on the Razor view engine to embed .NET code within HTML. Views should contain minimal logic, and the logic that is there should be related to presenting content.

ASP.NET WebForms vs. ASP.NET MVC:

ASP.NET WebForms are an outdated way of implementing the MVC design pattern into ASP.NET. The idea was to bring desktop-style application programming to the web. For example, clicking a button would fire an event that would get recieved by an event handler function. In ASP.NET WebForms, Views relied on an HTML <asp: /> tag and certain attributes to execute server-side code e.g. <asp:Button id=”button” runat=”server” OnClick=”DoSomething” />.

Unfortunately, ASP.NET WebForms were much to bloated for smartphones, thus we needed something that was lighter weight. ASP.NET MVC is less reliant on the web server to process user interactions. It also has a greater adherence to the MVC design pattern and absolute control of the rendered HTML.

ASP.NET MVC Core:

In the same move to make .NET available for all platforms with .NET Core, ASP.NET MVC Core moved the ASP.NET code to open source. ASP.NET MVC Core supports Windows, Mac, and Linux. It provides Razor pages for embedding .NET code into HTML, as well as Blazor, which allows you to use .NET code alongside JS. ASP.NET Core has support for Remote Procedure Call (RPC) using gRPC, a cloud-ready environment-based configuration system, lightweight, high-performance, and modular HTTP request pipeline and the ability to host on Kestrel, IIS, HTTP.sys, Nginx, Apache, and Docker.

ASP.NET MVC Core – Core Concepts:

Before I get too ahead of myself on the theory, it will help you to visualize the project structure of ASP.NET MVC if we create a project template. I’ve already covered how to install the .NET SDK near the beginning of this document. You should have the dotnet command-line utility at your disposal. In order to create an ASP.NET MVC project, create a project directory e.g. mkdir HelloWorld. Now, cd into the empty directory and run dotnet new mvc. This will generate the default layout for your ASP.NET MVC project. The project file templates already have a default page created for us, so execute dotnet run. The dotnet run command implicitely runs dotnet build so this should build the project, and then run it. The command will run in the foreground, so you won’t have control over your terminal. That’s okay though, because we only require the output of running this command. You should see at least one message displaying the localhost port that the application is running on. Depending on your setup, you may see two separate port numbers. One will be for HTTP and one for HTTPS. Either way, copy one of the links, paste it in your browser, and you should see a generic welcome page appear with your project name in the top left corner.

Now, let’s begin to explore the project structure. In the root of the project directory, there should be a .cs file called Program.cs. This file is where Main() is located. The file does not *necessarily* need to be called Program.cs, but this is the universal default. We will be revisiting this file, but essentially, it is in charge of handling all the setup of the application. It configures services that are required by the web app, creates the app’s request processing pipeline, and enables things like support for static files, routing, authorization, etc.

You’ll notice the first two lines in Main() are something like:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

var app = builder.Build();

These two lines initialize the hosting platform (web server). By default, ASP.NET MVC Core applications host themselves since they use Microsoft’s hosting platform, Kestrel. Kestrel is a cross-platform web server for ASP.NET Core that supports HTTPS, HTTP/2, WebSockets, Unix sockets for high performance on Nginx.

I briefly mentioned that Program.cs sets up services which are required by the web app. These are essentially just dependencies which get injected into the application. You would add them between the two lines that I highlighted above when initializing the web server (under the comment that says “Add services to the container”). That might look something like the following:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

builder.Services.AddScoped<SampleService>();

Var app = builder.Build();

You’ll also notice a line that checks if (!app.Environment.IsDevelopment()) { ... }. .NET has the ability to detect the environment that it’s running on. There are four types of environment you can check for: IsDevelopment(), IsProduction(), IsStaging(), and IsEnvironment(). The first three are basic checks for dev, prod, and staging. The fourth accepts a string parameter which is the environment name for a custom environment that you create.

launchSettings.json:

If you now cd into the Properties directory (located in the same directory level as Program.cs), you will see a file called launchSettings.json. This initializes the default launch settings for the application. If you recall how I told you to check the output of the dotnet run command for the localhost port, you will see in launchSettings.json that the port number that was chosen is actually statically defined here under the project build profile. You environment is also initialized here (set to Development by default), hence why app.Environment.IsDevelopment() will be true. This can also be changed on the command line e.g. export ASPNETCORE\_ENVIRONMENT=’STAGING’.

Static Files:

As is common with most MVC frameworks, static files such as .css, .js, libraries, and images, are stored under the wwwroot directory. This allows these resources to be accessed via a path which is relative to the web root e.g. https://<hostname>/images/<image\_file\_name>.

Model Class Example:

While were poking around in the auto-generated MVC project, let’s take a look in the Models directory. You’ll notice that one Model is defined, called ErrorViewModel.cs. This is a C# class file (revealed by the .cs extension). It contains a class that is named after the file (same as Java’s convention). The first line is a property called RequestId that defines getters and setters for that field. Note that the string reference type has a question mark after it, meaning that it is nullable. A lambda function is also created called ShowRequestID which returns true if RequestId is not null, otherwise it returns true.

Controller Class Example:

Next, let’s look at the Controllers directory where our controllers are stored. Another class file is located here called HomeController.cs. Notice that the naming convention for MVC files is to begin with the page name (in this case, Home), and then we append one of View, Controller, or Model, respectively e.g. HomeView.cshtml, HomeController.cs, HomeModel.cs. Anyhow, looking at this file, we see that the HomeController class extends Controller. As far as I’m aware, you must always extend the Controller class for MVC Controllers. A private, readonly field is created called \_logger which is of type ILogger<HomeController>. The constructor is doing something quite clever here. This is called dependency injection. We’ll cover dependency injection in the next section. Let’s skip past the constructor and look at the Index() and Privacy() methods. Methods which belong to a Controller class in ASP.NET are officially deemed “actions”, not methods; hence why the return an IActionResult. Notice that both of these actions just return View(), which is a method that returns a ViewResult (a class that is derrived from ActionResult).

The View() method is a tad bit tricky to grasp at first. It is a method that has multiple overrides. The default behavior of View() when there are no arguments passed to it is to return a View with the same name as the action method from which it is called. You can specify the view explicitely by passing in the name as a string e.g. return View(“About.cshtml”);. The argument here is actually a relative path, but we can make it an absolute path as well e.g. return View(“Views/Home/About.cshtml”);. Additionally, the file extension is optional. View() can also take a model as an argument e.g. return View(About); where About is an instance of an object that extends Model. Finally, View() has an override for returning both View and Model e.g. return View(“About”, About);

Dependency Injection:

A dependency within our code is an object that another object depends on. Let us first see what the problem actually is before I go on to show you how to solve it. Take the following code:

public class MyDependency

{

public void WriteMessage(string Message)

{

Console.WriteLine($”MyDependency.WriteMessage Message: {Message}”);

}

}

public class Dependent

{

private readonly MyDependency \_dependency = new MyDependency();

public void OnGet()

{

\_dependency.WriteMessage(“Dependent.OnGet created this message.”);

}

}

So what we’re doing here is creating two classes: MyDependency and Dependent. The Dependent class creates an instance of MyDependency called \_dependency which invokes it’s WriteMessage() method with an argument specific to the OnGet wrapper method.

The issue with this approach is that if the MyDependency class is modified, then the Dependent class will also need to be modified to match the changes. This is also difficult for unit testing as well. The solution is to use an interface or base class to abstract the dependency implementation. We’ve actually already seen dependency injection with services in the Program.cs file, as well as an implementation in the HomeController class. Here is the solution to the code example that I gave using dependency injection:

public interface ImyDependency

{

void WriteMessage(string Message);

}

public class MyDependency : ImyDependency

{

public void WriteMessage(string Message)

{

Console.WriteLine($”MyDependency.WriteMessage Message: {Message}”);

}

}

public class Dependent

{

private readonly IMyDependency \_myDependency;

public Dependent(IMyDependency myDependency)

{

\_myDependency = myDependency;

}

public void OnGet()

{

\_myDependency.WriteMessage(“Dependent.OnGet”);

}

}

public class Main

{

public void Main()

{

Dependent dep = new Dependent(new MyDependency());

}

}

That code may be difficult to read in something like libreoffice/MS Word, so I suggest copying it to an IDE/text editor. So the key differences here are that MyDependency now implements an interface and creates its own implementation of WriteMessage(). The other key difference is that now we have a constructor which accepts an IMyDependency as an argument, and copies that reference into its private readonly \_myDependency field. In the OnGet() method, the \_myDependency field invokes WriteMessage(), but the main point here is that the WriteMessage() method being invoked is an implementation of the WriteMessage() method defined in the IMyDependency interface. Now, in order to change the functionality of WriteMessage(), we simply create a second MyDependency class (with a different name of course) that also implements IMyDependency and creates its own implementation of WriteMessage(). Then, rather than change the Dependent class, all we need to do is switch the argument given to the Dependent class’ constructor in Main() to be an instance of MyDependency2 (or whatever name you give to the other MyDependency class).

Azure:

Going forward, I may mention Azure. Azure is Microsoft’s cloud services platform, similar to Amazon’s AWS, or Google’s Google Cloud services (or other, smaller cloud platforms, such as Linode or DigitalOcean). I think it’s important that we review cloud technologies, and why they are useful/necessary. You will sometimes hear the term “load balancing” or “load estimation”. Load refers to the amount of traffic on a given site (people accessing the site and making requests). Load balancing is accomplished with something called a load balancer, which is a server that directs internet traffic towards specific nodes in the network so that we do not overload a single node with requests. Load estimation is, well, estimating how much traffic your site will need to account for. This number tends to increase the longer that a site is up, thus we require a “scalable” solution to manage the amount of traffic as it increases. Cloud computing gives us a ton of computing services such as servers, storage, databases, networking, software analytics, etc. The nice thing about these services is that you only pay for what you use.

There are three types of cloud:

**Public cloud:** Owned by third-party cloud service providers e.g. Azure, AWS, GCP

**Private cloud:** Resources that are used exclusively by a single business or organization. Can be physically located on the company’s on-site data center, or they can be set up by other third-party service providers, so long as they are maintained on a private network.

**Hybrid cloud:** Public and private cloud combined.

There are also various cloud services that are offered by cloud service providers:

**On-Site:** You manage everything (applications, data, runtime, middleware, OS, virtualization, server, storage, networking).

**Infrastructure as a Service (IaaS):** They manage the low level infrastructure (networking, storage, servers, virtualization), you manage the high level stuff (OS, middleware, runtime, data, applications).

**Platform as a Service (PaaS):** They manage the low level infrastructure and the platform (networking, storage, servers, virtualization, OS, middleware, runtime), and you mange the application and data.

**Software as a Service (SaaS):** They manage everything, including your application. You trust them to do everything for you.

Recap of MVC:

What we’ve learned so far is that an incoming HTTP request will get routed to the Controller (more on routing in a bit). The Controller processes the request and creates a presentation Model and selects and appropriate View. The Model gets passed to the View so that it can access its data. The View will transform the Model into HTML tags to be rendered on the client side.

The View:

Views are located under the Views directory. Since there are often many more Views than there are Controllers or Models, it is more likely to see subfolders within our Views directory. View files end with the .cshtml extension rather than .cs. .cshtml files are better known as Razor files. Razor is a language that is nearly identical to C#, but functions more like PHP by embedding custom tags within our HTML. Here is a simple example of a Razor page:

@{

Layout = “\_Layout”;

ViewData[“Title”] = “Index”;

}

<h3>Hello World!</h3>

Razor code begins with the ‘@’ symbol. If you wanted to print a variable using Razor, you could simply do so without any code block e.g. <p>@var</p>. Anything that requires whitespace and is also mixed with HTML tags should be placed within a block like in the original example I gave. Speaking of which, let’s analyse the first code example. We specify a variable named Layout. This is a pre-defined variable (you can tell because we’re not assigning it any type identifier). We set it to the value “\_Layout”. This refers to the \_Layout.cshtml file which is located under Views/Shared/. This file is important because it is our primary layout. A layout is an HTML file that will typically contain content that you want to render on each and every page. It is quite common to see the page header and footer get put in the \_Layout.cshtml file since those usually get rendered on every page. For the body, we can use Razor to call a function called RenderBody() e.g. @RenderBody() which will insert the appropriate View in the body (between the header and footer). How do we know which View will get put in the body? Well, based on the URL that we enter. The URL will contain our domain, followed by the action that we want to invoke. Remember that an action is just a method within our Controller that returns an ActionResult. The View() method returns a ViewResult which is a class derrived from ActionResult. So, the action that we pick in the URL dictates which action gets invoked, and then that action may or may not return View(); which returns the appropriate View to get rendered in @RenderBody(). Seems confusing, but it’s not that bad after you write some code.

To recap on layouts: Layout files contain HTML code that we want to repeat across multiple pages (header, footer, nav bar, etc.). Applications can have zero, one, or many layouts. The default layout created by ASP.NET MVC Core is \_Layout.cshtml which is located in the Views/Shared/ directory. Layouts can invoke @RenderBody() to render a **maximum** of *one* View! Something called a partial view exists, which renders on top of the primary View, but is not an entire View in and of itself. Partial views can be rendered with the @RenderSection() function.

View Directives:

Common directives that are to be used in multiple Views are included in the \_ViewImports.cshtml file. This file is located in the Views directory. \_ViewImports.cshtml supports the following directives:

**@addTagHelper:** Makes tag helpers available to a View

**@removeTagHelper:** Removes tag helpers previous added from a View

**@model:** Specifies the type of Model passed to a View or page

**@inherits:** Controls the interface(s) that the View inherits from

**@using:** Adds the C# “using” directive to the generated View e.g. @using System.IO

**@inject:** Enables the Razor page to inject a service from the service container into a View

In the \_ViewImports.cshtml file that is autogenerated by the dotnet command line utility, we see that it has two @using directives which specify the namespace that was created for the project, as well as the Models sub-namespace. We add all tag helpers using @addTagHelper \*, as well as the ASP.NET Core MVC tag helpers (Microsoft.AspNetCore.Mvc.TagHelpers). Note that \_ViewImports.cshtml gets applied to all Views within the same directory as well as all subdirectories.

Pre-Processing:

Pre-processing can be done in a neighboring file to \_ViewImports.cshtml called \_ViewStart.cshtml. This file gets run before a full View is loaded (this does not include layouts or partial views). Similar to \_ViewImports.cshtml, \_ViewStart.cshtml gets applied to all Views within the same directory as well as any and all subdirectories.

Partial Views in Detail:

I briefly covered partial views, but how do they work, exactly? Well, same as regular Views, they are Razor markup files (.cshtml) which render HTML content within a regular View. An example use-case would be widgets. Each widget could recieve its own partial view to be rendered on top of the main View. We can directly load a partial view within a main View using the @Html.Partial() function. The argument will always be a string, however, it can be written using a relative or full path. Here are examples:

// Searches the same directory as the View. If not found, searches Shared directory

@Html.Partial(“ViewName”)

// Partial view MUST be located in the same folder as the View

@Html.Partial(“ViewName.cshtml”)

// Locate partial view based on full path. Boht “~/” and “/” refer to the application root directory

@Html.Partial(“~/Views/Folder/ViewName.cshtml”)

Razor:

Even though we’ve discussed Razor at this point, I did not go into much detail about it. Valid Razor types include int, float, decimal, bool, and String. This is probably the biggest difference between it, and C#. Here is an example of generating dynamic content using Razor:

<html>

<body>

@for (var i = 0; i < 10; i++)

{ <p>Count @i</p> }

</body>

</html>

This, as you can probably guess, outputs the numbers 0-9 on your page. As you can probably imagine, Razor can do things like switch statements, conditional branches (if, else if, else), while loops, etc.

ASP.NET MVC Forms:

Now we really begin to delve into the interaction between the client-side application and our backend. Hopefully, you are familiar with HTML forms using the <form> tag and <input> tags. A form collects all data from the input fields and then does a PUT/POST to the server. Don’t forget that <input> tags accept a “type” attribute which can be one of many values e.g. email, number, password, submit, tel, time, url, etc.

ASP.NET MVC forms are identical to regular HTML forms and they use the same underlying technology. Remember: HTML forms != ASP.NET WebForms, which we covered earlier. The way that we create an ASP.NET MVC form is by adding the asp-controller and asp-action attributes to our <form> tag e.g.

<form asp-controller=”Home” asp-action=”DisplaySession” method=”post”></form>

Any data that is submitted on this form will make its way to whatever View gets invoked by the DisplaySession action in HomeController.cs.

View Types:

There are essentially three methods of coupling our Views with our Controllers. These methods are:

- Strongly typed views

- Weakly typed view

- Dynamic views

A strongly typed View is a View that is bound to a Model. This is the preferred method of transferring data from the Controller to the View. To do this, we use the @model directive in our View to specify the Model that we want to bind. In the Controller, we must invoke the View() method with the model as a parameter. Where do we get the Model object? It will be automatically provided to us as an argument to our action; therefore, we must include it as a parameter. For example, in our Controller, we add:

[HttpPost]

public IActionResult DisplayForm(LoginInformationModel model)

{

return View(model);

}

In our View, we add:

@model ProjectName.Models.LoginInformationModel

@{

Layout = “\_Layout”

ViewData[“Title”] = “Display Form”;

}

<h3>Display Form Data</h3>

<table>

<tr>

<td>

Email Address:

</td>

<td>

@Model.EmailAddress

</td>

</tr>

...

</table>

Note the @model directive used at the top, which allows us to access the Model object. The Model object will create a copy of the Model reference that was passed to DisplayForm().

For weakly typed Views, the View is not bound to a model. The Controller must pass the data to the view using one of the following: DataBag, ViewData, TempData, or Session Data. This is useful if we are only sending a small peice of information to the View that does not need to be associated with a Model. Here is the same example as before, but using weakly typed views:

[HttpPost]

public IActionResult DisplaySession()

{

HttpContext.Session.SetString(“emailAddress”, Request.Form[“emailAddress”]);

HttpContext.Session.SetString(“password”, Request.Form[“password”]);

return View();

}

In the View:

@using Microsoft.AspNetCore.Http;

@{

Layout = “\_Layout”;

}

<h3>Display Session Data</h3>

<table>

<tr>

<td>

Email Address:

</td>

<td>

@Context.Session.GetString(“emailAddress”)

</td>

</tr>

...

</table>

The @Context.Session object contains our session data, but recall that I said we could also use ViewData or ViewBag as well. Thus, we can replace @Context.Session.GetString(“emailAddress”) with either @ViewData[“emailAddress”] or @ViewBag.emailAddress. Here is the revised Controller to demonstrate this:

[HttpPost]

public IActionResult DisplayViewData()

{

ViewBag.emailAddress = Request.Form[“emailAddress”];

ViewData[“password”] = Request.Form[“password”];

return View();

}

ViewData is a dictionary, which is why we use the index operator ([]), unlike ViewBag, which uses the dot operator. A dictionary is basically a hash map that contains key-value pairs. In our Controller, we get our form data using Request.Form, which is also a dictionary, and returns the values associated with the keys that we supply on the form that was PUT/POSTed. In terms of the differences between ViewData, ViewBag, TempData, and Session, I will quickly explain.

**ViewData:**

- Is derived from ViewDataDictionary

- Its life cycle only lasts during the current request

- If redirection occurs, it becomes null

- Requires type cast

**ViewBag:**

- Is a wrapper around ViewData

- Its life cycle only lasts during the current request

- If redirection occurs, it becomes null

- Does not require type cast

**TempData:**

- Dictionary object derived from TempDataDictionary

- Used to pass data from current request to subsequent request (pass data when redirecting from one page to another)

- Life cycle is very short and lies only until the target View is fully loaded

- Typecasting is required

- Stores things like one-time messages, error messages, validation messages

**Session:**

- Persists for a set expiration time (default 20 minutes)

- Session is valid for all requests, not just a single redirect

- Requires type casting

Finally, let us discuss dynamic Views. A dynamic View is not bound to a Model. However, data is still passed to the View via a Model instance. This is the least preferred method of data transfer. Here is the Controller example:

[HttpPost]

public IActionResult DisplayDynamicFormData()

{

var res = Request.Form[“personDropDown”].ToString();

if (res.Equals(“Student”))

{

return DisplayDynamicStudent();

}

else

{

return DisplayDynamicTeacher();

}

}

[HttpPost]

public IActionResult DisplayDynamicStudent()

{

return View(“DisplayDynamic”,

new Student

{

Name = “John”,

StudentId = “1”

});

}

Above, we check the form that was returned via a PUT/POST request on the client side and check if there were any values called “Student”. If the value *is* “Student”, we invoke the DisplayDynamicStudent() action, otherwise, we invoke the DisplayDynamicTeacher() action. In DisplayDynamicStudent(), we pass both a View and Model to the View() method. The Model we create/initialize on the fly, and give it the arguments “John” and “1” for the Name and StudentId respectively. Here is the View:

@{

Layout = “\_Layout”;

ViewData[“Title”] = “Dynamic”;

}

<h3>Dynamic View</h3>

<form asp-controller=”Home” asp-action=”DisplayDynamicFormData” method=”post”>

<table>

<tr>

<td>Choose One:</td>

<td>

@Html.DropDownList(“PersonDropDown”, new List<SelectListItem>

{

new SelectListItem{text=”Student”, value=”Student”},

new SelectListItem{text=”Teacher”, value=”Teacher”}

})

</td>

</tr>

</table>

<input type=”submit” value=”Select” />

</form>

Here we use the Html.DropDownList function to create a new drop down menu called PersonDropDown. We initialize it with two items: Student, and Teacher. If the user selects the Student item, the Controller invokes DisplayDynamicStudent() which returns the DisplayDynamic view with John being the only student that will get rendered. Otherwise, if the Teacher drop down item is selected, DisplayDynamicTeacher() gets invoked, which would presumably do the same thing as the student view, but for teachers.

Routing:

Now we come to one of the biggest topics in ASP.NET: Routing. Routing is a common method used in web applications to perform actions or return content based on the contents of the URL. To understand routing, we’ll start by looking at the ASP.NET MVC Core Request Pipeline (a real mouthful).

The Request Pipeline is sort of like a stack of request delegates; a request delegate being a function pointer/reference to a method. Each delegate is referred to as Middleware. Thus, we can say that Middleware is software that is assembled into an application’s pipeline to handle requests and responses. The Request Pipeline is invoked upon every request to the application. Each Middleware in this pipeline chooses whether or not to pass the request to the next component in the pipeline. Each Middleware can perform work before and after the next component in the pipeline.

Let’s go back to our Program.cs file, as I promised we would. Note the lines that begin with app:

app.UseRouting();

app.UseStaticFiles();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: “default”,

pattern: “{controller=Home}/{action=Index}/{id?}”);

});

app.UseRouting() adds route matching to the Middleware pipeline. This Middleware looks at the set of endpoints defined in the app and selects the best math based on the request. app.UseStaticFiles() allows the app to be able to discover and show static files e.g. images. Finally, and most importantly, is UseEndpoints() which adds endpoint execution to the Middleware pipeline. We will come back to this shortly.

Defining Middleware:

This is about to get confusing again (sorry). We can create our own Middleware to add to the Request Pipeline. There are three kinds of Middleware that we can create: chaining, branching, and terminal Middleware. We use the Use(), Map(), and Run() functions to define chaining, branching, and terminal Middleware, respectively. By chaining Middleware, I mean Middleware that will invoke the next Middleware in our “linked list”. Branching Middleware is essentially an if conditional. Finally, Run() is a terminal Middleware, which halts the entire pipeline, and no further Middleware is run. Here are examples of creating chaining, branching, and terminal Middleware:

app.Use(async (context, next) =>

{

// Do work that doesn’t write to the response

await next.Invoke();

// Do other work that doesn’t write to the response

});

app.Map(“/home/action”, HandleAction);

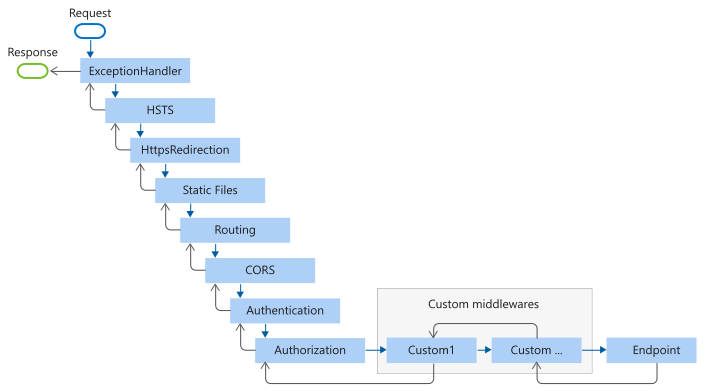
app.Run(async, context =>

{

await context.Response.WriteAsync(“Hello World!”);

});

Middleware Execution Order:

 Middleware gets executed in the order that you define it in the Program class. Order is important for security and functionality.   
  
  
  
  
  
  
  
  
  
  
  
  
  
If you look at this chart long enough (or if you just read what I’m telling you right now), you’ll notice that we recieve a request (the blue elipse) and we dig down the pipeline following the same blue arrow towards the endpoint. But note that for every request, we must travel back through the pipeline to send a response. Hence the grey array which back-tracks up from the endpoint, back to the response.

Routing Continued:

Getting back to routing, it is essentially what allows us to navigate through our application. It is responsible for matching incoming HTTP requests and dispatching them to the application’s executable endpoints. There are many different ways to configure routing in .NET. Some methods include Razor pages, Controllers, SignalR, Remote Procedure Call (RPC) handled in .NET by gRPC, Endpoint-enabled Middleware e.g. Health Checks, delegates and lambdas registered with routing.

We require routing because ASP.NET MVC Core applications no longer use static files and locations to display their content. Therefore, each “route” points to a pre-defined, pre-compiled location, and not necessarily to a file. For example, <https://www.myapplication/home/index> might point to a controller action, and then a view will be rendered by the server in a Just-in-Time (JIT) manner.

Each “route” specifies which Controller should be accessed, which View should be rendered, and what data should be passed between locations. This is technically not the full picture, because routes can do more than just invoke actions, but displaying content and passing arguments will be our main focus for now. Let’s take a look at setting up routing in Program.cs again:

app.UseRouting(); // Adds route matching to the Middleware pipeline

// UseEndpoints invokes the delegate associated with the selected endpoint

app.UseEndpoints(endpoints =>

{

// MapGet(“/”) == if (GET request for root page) { ... }

endpoints.MapGet(“/”, async context =>

{

await context.Response.WriteAsync(“Hello World!”);

});

}

This peice of code will return the text “Hello World!” if we do a GET request on the root page of the application. For example, trying to access <https://localhost:5001/> with a POST request would return a 404 error, since endpoints.MapGet is checking that the request was a GET request. Similarly, a GET request on <https://localhost:5001/home> would also fail, since we specifically setup the routing to display the text when accessing the root of the application. Therefore, in order to have the text appear, we would need to do a GET request on <https://localhost:5001/>.

The issue with using MapGet() for everything is that it is a lot of work to define a URL pattern for every single action in our application. And even if we could, we’d still have to find a way to handle the order of execution have error handling. The solution is to use one of conventional routing i.e. routing through actions in Controllers, or attribute routing i.e. routing through Controllers using attribute decorators (used for web APIs).

Conventional Routing:

Let’s once again analyze the Program class. I’ve already highlighted this before, but now we’ll take a second look. You’ll notice this in your auto-generated MVC project:

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: “default”,

pattern: “{controller=Home}/{action=Index}/{id?}”

);

});

This time, instead of MapGet(), we use MapControllerRoute(). MapControllerRoute() implements conventional routing. The pattern describes how we route based on the contents of the URL. The pattern matching string follows a specific “syntax”. The curly braces encapsulate pseudo code that will be executed. controller, action, and id are all variables, the first two of which, we are assigning default values. The slashes between all of these map to sections of the URL. So a URL such as /Hello/World/1 will assign the value Hello to controller, World to action, and 1 to id. The id gets passed as an argument to the action e.g.:

public class HelloController : Controller

{

public IActionResult World(int id)

{

return View();

}

}

Note that the ‘?’ after id is a special character that denotes its existence being optional. Therefore, we do not *have* to include it in the URL. Similarly, we can add an asterisk before a variable in the pattern matching string as a wildcard. For example:

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute (

name: “blog”,

pattern: “blog/{\*article}, // Matches anything that comes after blog/

defaults: new { controller = “Blog”, action = “Article” }

);

endpoints.MapControllerRoute (

name: “default”,

pattern: “{controller=Home}/{action=Index}/{id?}”

);

}

We can even apply routing constraints e.g. blog/{article:minlength(10)} which would suggest that the section following blog/ would need to be at minimum 10 characters long. Here is a summary:

- Tokens within {} define route parameters

- If number of route parameters > 1, route parameters must be separated by ‘/’

- Route parameters must have a name

- Text matching is case-insensitive

- The ‘=’ assigns a default value to our parameters

- The escape character for route templates is ‘{‘ or ‘}’ e.g. “{{“ would evaluate to ‘{‘

- ‘\*’ is a catch all and doesn’t replace ‘/’. “\*\*” is a catch all and replaces ‘/’

- Optional parameters are marked with ‘?’

- Parameters can have constraints e.g. {article:minlength(10)}

On a final note for conventional routing, note that it executes in sequence, meaning that if there are multiple endpoints that use MapControllerRoute() and some of them use catch-alls with the kleene star (\*), then those may become greedy routes and match with URLs that were intended to match with routes defined later down the list. The best solution at the moment is simply to put patterns that use the kleene star closer to the end of your list of endpoints so that they get matched at the end.

REST Attribute Routing:

I mentioned earlier that there were two types of routing we could use: conventional routing (which we just covered) and attribute routing (which we will cover now). I briefly mentioned that attribute routing is used with web APIs, i.e. RESTful APIs. Rather than setup a pattern matching string in our Program class, attribute routing is performed within the Controller. To use attribute routing, we must use the MapControllers() endpoint in app.UseEndpoints e.g.:

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

The idea behind attribute routing is that we model the application’s functionality as a set of resources where operations are represented with HTTP verbs (GET, PUT, POST, DELETE, etc.) There are a few reserved routing words that we can use within our Controllers. These include: action, area, controller, handler, and page. This might look something like the following:

[Route(“[controller]/[action]”)]

public class HomeController : Controller

{

[Route(“~/”)]

[Route(“Home”)]

[Route(“~/Home/Index”)]

public IActionResult Index()

{

return ...

}

public IActionResult About()

{

return ...

}

}

Notice the similarity between our pattern matching string with conventional routing and the route defined prior to the class declaration. Notice that that string uses two of the keywords that I mentioned we were allowed to use in the Controller (the keywords being controller and action). These indicate that the expected URL must contain the controller parameter and action parameter under normal circumstances. The other routes defined about Index() are exceptions to this rule. ~/ indicates that if we access the root of the application that Index() will still be invoked. Similarly, adding Home to the URL or ~/Home/Index will all invoke the Index() method.

Routing With HTTP Verbs:

Another way of doing attribute routing is to use HTTP verbs. These use the same pattern matching strings as conventional routing, but add HTTP verbs. Here are some examples:

- [HttpGet]: Only gets invoked if the route matches and the request is a GET request

- [HttpGet(“{id}”)]: Same as prior, but appends a non-optional id to the end of the URL

- [HttpGet(“int/{id:int}”)]: Same as prior, but appends non-optional integer type to the URL

- [HttpPost]: Only gets invoked if the route matches and the request is a POST request

Route Template Precedence:

In terms of precedence when using attribute routing, there are a few rules to remember:

- Templates with more segments (parts between ‘/’ characters) are considered more specific

- A segment with literal text e.g. [(“/home”)] is considered more specific than one without

- A complex segment e.g. [(“/a{b}c{d}”)] is considered as specific as a parameter segment with a constraints

- Catch-all parameters e.g. [“/home/{\*\*blog}”] are the least specific. See catch-all in the Route template reference for important info on catch-all routes

Note: These rules apply to conventional routing as well

Conventional Routing or Attribute Routing?:

The suggestion is to use conventional routing for Controllers serving HTML pages for browsers, and attribute routing for Controllers serving REST APIs. Actions are typically either conventionally routed or attribute routed, not both. If you place any routes in the Controller, it immediately makes all actions in the Controller attribute routed.

Terminal Middleware vs. Routing:

Both Middleware and conventional/attribute routing allow terminating the process pipeline. Middleware terminates the pipeline by returning rather than invoking the next Middleware. Endpoints are always terminals. Terminal Middleware allows positioning the Middleware at an arbitrary place in the pipeline. Endpoints are executed at the position of UseEndpoints(). Terminal Middleware allows arbitrary code to determine when the Middleware matches. Custom route matching code can be verbose and difficult to write correctly. Routing provides straightforward solutions for typical applications. Most apps don’t require custom route matching code. Endpoints interface with other Middleware but using a Terminal Middleware with other Middleware requires manual interfacing.

Routing from Views Using Tag Helpers:

We briefly mentioned tag helpers when discussing the directives that are allowed in \_ViewImports.cshtml. Tag helpers are used to add metadata to already existing HTML tags. They are considered to be an add-on library to the MVC Core framework. They help to simplify workflow/development and must be used in conjunction with a field called in \_ViewImports.cshtml. We’ve actually seen tag helpers be used in our ASP.NET forms before. Here is the syntax:

Basic Route:

<a asp-controller=”Home” asp-action=”RouteView1”>Go to Home/RouteView1</a>

Route With Parameter:

<a asp-controller=”Home” asp-action=”RouteView2” asp-route-id=”42”>Go to Home/RouteView2 Passing Id=42</a>

Route With Multiple Parameters:

<a asp-controller=”Home” asp-action=”RouteView3” asp-route-firstName=”Neil” asp-route-lastName=”Kingdom”>Go to Home/RouteView3 Passing firstName=Neil and lastName=Kingdom</a>

Model Binding:

Since we’re on the topic of ASP.NET forms, let’s review how form data gets sent to the Model. For instance, if I have an action e.g.:

public IActionResult DisplayForm(LoginInformationModel model)

{

return View(model);

}

How do the values submitted on the form get injected into my model’s member variables? This is handled by something called the Model binder, which is responsible for populating the action method parameter. The Model binder retrieves data from various sources, such as route data (e.g. the optional id field we had), form fields, and query strings e.g. when you place a ‘?’ at the end of the URL and set certain parameters (https://hostname:port/Home?FirstName=Neil&LastName=Kingdom). The Model binder is responsible for taking data from all of these sources, converting the string data to .NET types, and updating properties of the Model.

ASP.NET Core grants several attributes which give finer grain control over which sources we accept binding data from. Here is a list:

[FromForm]

[FromRoute]

[FromQuery]

[FromBody]

[FromHeader]

[FromServices]

If the Model binder fails to bind the incoming request data to the corresponding Model property, then it does not throw any errors, but fails silently. The above applies to Controllers with Views. We will see in the future that this rule is slightly different for web APIs. In order to check if Model binding has succeeded, we can check the ModelState.isValid attribute.

Entity Framework:

Alongside routing, the Entity Framework is a very important concept to grasp if we want to be able to have persistent state in our data. I’m talking about data storage on a DataBase, to be specific. The Entity Framework (EF) is a relational database mapping software build for .NET and .NET Core. The EF provides the programmer with an easy-to-use interface to a database and its “entities” which are accessed via a data context. The term Entities refers to tables, views, stored procedures, etc. The data context refers to the programmer interface. To summarize, EF is a code-less CRUD web interface. EF Core is the latest version of the Entity Framework and completely rewrites it from the ground up. Entities usually map to Models in MVC applications.

What is an Object Relational Mapper (ORM)?:

ORMs are used to create a “code version” of your database and vice versa. Essentially, tables in your DB are turned into Model classes called Entities. As I already mentioned, Entities can be tables, views, stored procedures, indexes, etc. As developers, we mostly work with the application business objects and the ORM framework generates the required SQL. ORMs also create a programming interface called a data context. Note that EF is just one ORM technology for .NET, but others, such as LINQ to SQL and NHibernate, also exist.

Without an ORM such as EF, in order to communicate with a DB, you are required to create a “data access layer”, which is a peice of code that takes each and every entity class from .NET and converts it into SQL queries (and vice versa). This is a huge pain, because anytime you make a change to an entity class in .NET, you need to not only change the code in your data access layer, but also change you database as well. Everything is handled manually, which is simply not a good work flow.

With EF, we have two possible solutions to this problem. The two approaches I’m speaking of are called “code-first” and “database-first”. You may be able to guess how each of these function. With code-first, we write the code before creating the DB. We write the application entity classes, create a DbContext object (more on this in a bit), and then EF Core uses a process called migration to create the DB and tables based on the DbContext and your entity classes. The pros of this approach are that code is easier to write than creating an entire DB. There is less auto-generated code, and you can version your database i.e. rollback your database by simply reverting to a previous commit in Git and then re-generating the tables in the DB. The downside to this approach is that everything about the DB needs to be written in code. This is easy enough for tables, but less convenient for advanced topics such as stored procedures, triggers, etc.

In terms of database-first, this is useful primarily if you already have a working database. EF Core can create the DbContext and Entity classes based on the DB schema. The pros of this approach are that the database is already ready, so you can immediately begin using it. Databases also support a lot of advanced features which are not supported by EF. The cons are that this creates a lot of autogenerated code which is difficult to manage, and if the DB changes, you need to regenerate the code. For obvious reasons, this approach is not typically preferred over code-first.

Migration:

I briefly touched on migration a moment ago. It is the process of keeping the database in sync with Entities within your code.

// TODO: add linux command to add ef migrations

When you create a migration, the framework compares the current state of the Model with the previous migration or initial context and generates a class with an up method (contains code to apply changes to DB) and a down method (has code to revert the DB to the state before migration).

DbContext:

EF provides facilities to materialize data returned from the DB as entity objects. It can track changes that were made to those objects, handle concurrency, propogate object changes back to the DB, and bind objects to controls. The primary class responsible for interfacing with data as objects is DbContext. Using DbContext, tables (entities) are accessed through query-able collections i.e. Icollection<T>. In order to use DbContext in your application, you need to create a class that derives from the DbContext class e.g. public class FooDataContext : DbContext

{

var context = new FooDataContext();

var fooItem = (from f in context.Foo where f.Bar == “Baz” select f).FirstOrDefault();

}

DbSet:

DbContext represents the collection of entities in the context with the aid of DbSets. Including a DbSet of a type on your context means that it is included in EF Core’s model. We usually refer to such a type as an entity. The DbContext class includes a DbSet for each entity in your application. Without DbSet, the connection between the DbContext and the DB table is not created. E.g. public DbSet<Driver> Drivers { get; set; }

DbContex’s Lifetime:

The lifetime of a DbContext object is usually a single unit of work. This explains why we create a scoped service in Startup for it: services.AddDbContext<DrivingDbContext>(opts...); A typical unit of work when using EF involves: creation of a DbContext instance, tracking of entity instances by the context (entities become tracked by being returned from a query or being added or attached to the context), changes being made to the tracked entities as needed to implement business rules, SaveChanges() or SaveChangesAsync() is called, or the DbContext instance is disposed.

Role of DbContext:

- Manage DB connection

- Configure Model & relationship

- Querying DB

- Saving data to DB

- Configure change tracking

- Caching

- Transaction management

Notes on DbContext:

EF Core does not support multiple asynchronous operations being run on the same DbContext instance. This is for a variety of reasons – primarily the fact that if two asynchronous updates are made on the same field in the entity, we can’t know which one to apply since asynchronous operations have no method of identifying order of operations. You should always await async calls immediately after they are made.

DbContextOptions:

Another object called DbContextOptions is required to configure DbContext. Without DbContextOptions, DbContext cannot function. The instance of DbCOntextOptions carries all of the required configuration info such as the connection string, database provider, etc. e.g.:

public SchoolContext(DbContextOptions<SchoolContext> options) : base(options) { ... }

You generate and pass options in your program class. This is just one of three ways to build the context options. MSDN covers the other methods.

Connection Strings:

You require a connection string to connect to the DB. They need to be protected as they usually contain username and passwords. Connection strings can be stored in appsettings.json, environment variables, secret password store, etc.

Creating and Configuring the Models:

EF Core uses a set of conventions to build a Model based on the shape of your entity classes. You can specify additional configuration to supplement and/or override what was discovered by convention. There are two sets of tools to achieve this: Fluent API and Data Annotations. The Fluent API allows us to override the OnModelCreating() method in your entity class and edit certain properties about the entity. Data annotations are a bit more concise, but don’t provide the same amount of complexity as the Fluent API. Here is an example in Fluent:

class MyContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>().Property(b => b.Url).IsRequired();

}

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

}

}

And with data annotations:

class MyContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

}

public class Blog

{

public int BlogId { get; set; }

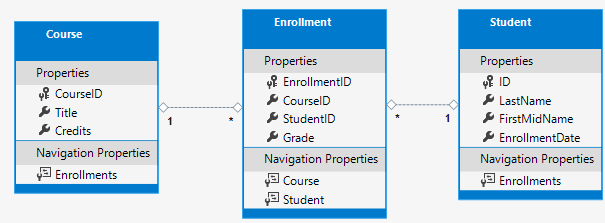
[Required] // Data annotation

public string Url { get; set; }

}

Database Review:

It’s been quite some time since I last reviewed DB theory (or since I’ve even touched a databse). Perhaps you are in the same boat. Well, in case you forgot, relational databases use various types of keys to create relationships. A primary key (PK) is expected to exist as the first column in every table. The PK can either be called ID or it can be prepended by the name of the table e.g. StudentID. Recall that we are not able to have many to many relationships between tables. The way that we solve a many to many relationship is by introducing an intermediate table that stores two foreign keys. Each foreign key (FK) maps to the corresponding PKs of the original two tables. Essentially, we’ve now created two 1:M relationships. We will take the example from Microsoft’s website and demonstrate how to implement it in code. Assume you have created the following database diagram:

  
Note that enrollment was created as a bi-product of the many to many relationship between Course and Student (many courses can have many students). CourseID and StudentID are FKs that were “imported” from Course and Student, respectively. Naturally, we begin by creating all three entity classes. We’ll begin with Student. In the Models folder, we’ll create a Student.cs file and create a Student class:

using System;

using System.Collections.Generic;

namespace ContosoUniversity.Models

{

public class Student

{

public int StudentID { get; set; }

public string LastName { get; set; }

public string FirstMidName { get; set; }

public DateTime EnrollmentDate { get; set; }

public ICollection<Enrollment> Enrollments { get; set; }

}

}

The most important thing in the code above is the Enrollment property. This is called a navigation property, and it holds other entities which are related to the current entity (Student). Thus, it contains all of the Enrollment entities related to Student. The navigation property must be a list such as ICollection<T>, List<T>, or HashSet<T>. Next, we create the Enrollment entity class:

namespace ContosoUniversity.Models

{

public enum Grade { A, B, C, D, F }

public class Enrollment

{

public int EnrollmentID { get; set; }

public int CourseID { get; set; }

public int StudentID { get; set }

public Grade? Grade { get; set; }

public Course Course { get; set; }

public Student Student { get; set; }

}

}

In this class, EnrollmentID is the PK, whereas CourseID and StudentID are FKs. Recall that since we have a 1:M relationship, an instance of Enrollment can only contain one reference to Course and Student, respectively, hence the Course and Student properties. Grade being nullable indicates that it may or may not be set (students will have null for their grade until the end of the semester). Next, we’ll create our Course entity:

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models

{

public class Course

{

[DatabaseGenerated(DatabaseGeneratedOption.None)]

public int CourseID { get; set; }

public string Title { get; set; }

public int credits { get; set; }

public ICollection<Enrollment> Enrollments { get; set; }

}

}

The only new thing here is the DatabaseGenerated attribute. Passing it an argument of DatabaseGeneratedOption.None means that the primary key will be provided by the user rather than generated by the DB since, by default, EF Core assumes that primary key values are generated by the DB. This is usually what you want, however, for Course entities, we assume a user-specified course number e.g. 1000, 2000, 3000, whereas autogenerated IDs will increment starting at 0, 1, 2, ... Finally, we must create the database context class. We will create a new directory in the root of our project called Data, which will contain SchoolContext.cs. Here are the contents:

using ContosoUniversity.Models;

using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data

{

public class SchoolContext : DbContext

{

public SchoolContext(DbContextOptions<SchoolContext> options) : base(options) {}

public DbSet<Course> Courses { get; set; }

public DbSet<Enrollment> Enrollments { get; set; }

public DbSet<Student> Students { get; set; }

}

}

I don’t think I mentioned this before, but the ‘:’ the proceeds the constructor/method signature for SchoolContext() functions the same way that it does in C++. Namely, the code the proceeds ‘:’ gets invoked before the method body. base() is equivallent to a call to super() in Java. It just invokes the constructor of the base class (DbContext in this case). A DbSet must be created for each entity in our DB. Each instance belonging to the set would be considered a row of that table, and the properties of that instance would be the columns. A silly, but important detail is that property names for collections are typically plural, but developers disagree on whether or not they should be plural or singular. Here is how you would override the default behaviour and make the tables singular:

using ContosoUniversity.Models;

using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data

{

public class SchoolContext : DbContext

{

public SchoolContext(DbContextOptions<SchoolContext> options) : base(options) {}

public DbSet<Course> Courses { get; set; }

public DbSet<Enrollment> Enrollments { get; set; }

public DbSet<Student> Students { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Course>().ToTable(“Course”);

modelBuilder.Entity<Enrollment>().ToTable(“Enrollment”);

modelBuilder.Entity<Student().ToTable(“Student”);

}

}

}

Notice that we’re using the Fluent API and overriding OnModelCreating() to cast our entities into tables with singular names. This could also be accomplished using data annotations to manually configure the table name. To do that, you’d put the following annotations above the SchoolContext class:

[Table(“Course”)]

[Table(“Enrollment”)]

[Table(“Student”)]

...

To round this all off, we must use dependency injection to include a database service in Program.cs. In Program.cs, under the comment that says “Add services to the container”, add the following: builder.Services.AddDbContext<SchoolContext>(options => options.UseSqlServer(Configuration.GetConnectionString(“DefaultConnection”)));

Normally, the connection string is passed to the context by calling a method on a DbContextOptionBuilder object. For local development, however, the ASP.NET Core configuration system reads the connection string from appsettings.json in the root of your project directory. You must add the following directly after the first opening brace ‘{‘ in appsettings.json:

“ConnectionStrings”: {

“DefaultConnection”: “Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted\_Connection=True;MultipleActiveResultSets=true”

},

Query Data:

EF using LINQ/Lambda to query data. Here are some examples:

// Query all

var blogs = context.Blogs.ToList();

// Query where BlogId == 1

var blog = context.Blogs.Single(b => b.BlogId == 1);

// Query all who’s Url fields contain “dotnet”

var blogs = contextBlogs.Where(b => b.Url.Contains(“dotnet”)).ToList();

// Eager loading: Load the data eagerly using joins

var blogs = context.Blogs.Include(b => b.Posts).ThenInclude(p => p.PostTags).ToList();

// Explicit loading: Explixitely specify what data to load

var blog = context.Blogs.Single(b => b.BlogId == 1);

var posts = context.Entry(blog).Collection(b => b.Posts).Load();

// Lazy loading: The related entities will not load until required (add to startup DbContext)

b => b.UseLazyLoadingProxies()

Saving Data:

// Insert

var blog = new Blog { Url = “[http://example.com](http://example.com/)” };

context.Blogs.Add(blog);

context.SaveChanges();

// Update

var blog = context.Blogs.First();

blog.Url = “<http://example.com/blog>”;

context.SaveChanges();

// Delete

var blog = context.Blogs.First();

context.Blogs.Remove(blog);

context.SaveChanges();

Asynchronous Programming:

Asynchronous operations avoid blocking a thread while the query is executed in the database. This is not quite the same as multithreading. In multithreading, you spawn multiple threads to do one task. In asynchronous programming, you set an operation to be asynchronous and then that operation returns whenever it’s completed. The main thread never halts execution. We’ll create a program that “makes breakfast” synchronously and then a few times asynchronously to demonstrate. First, the synchronous solution:

static void Main(string[] args)

{

Coffe cup = PourCoffee();

Egg eggs = FryEggs(2);

Bacon bacon = FryBacon(3);

Toast toast = ToastBread(2);

ApplyButter(toast);

ApplyJam(toast);

Juice oj = PourOJ();

Console.WriteLine(“Breakfast is ready!”);

}

private static Toast ToastBread(int slices) =>

{

Console.WriteLine(“Toasting...”);

Task.Delay(1000 \* slice).Wait();

return new Toast();

}

Now asynchronously:

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

Egg eggs = await FryEggsAsync(2);

Bacon bacon = await FryBaconAsync(3);

Toast toast = await ToastBreadAsync(2);

ApplyButter(toast);

ApplyJam(toast);

Juice oj = PourOJ();

}

private static async Task<Toast> ToastBreadAsync(int slices)

{

Console.WriteLine(“Toasting...”);

Task.Delay(1000 \* slices).Wait();

return new Toast();

}

Note that in ToastBreadAsync(), we mark the method as asynchronous using the async keyword, and we return a Task with generic type Toast, rather than an instance of Toast. The await keyword means that we are awaiting the task to complete. Eggs, bacon, and toast will only be initialized once their function calls return, however, we do not halt execution for any of our asynchronous operations. In order to have “true concurrency” i.e. executing all three of these tasks in parallel, the best way is to create a list of tasks that we want to wait for, and then await all of those tasks in a blocking loop. Example:

static async Task Main(string[] args)

{

Coffee cup = PourCoffee();

var eggsTask = FryEggsAsync(2);

var baconTask = FryBaconAsync(3);

var toastTask = MakeToastWithButterAndJamAsync(2);

var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };

while (breakfastTasks.Count > 0)

{

Task finishedTask = await Task.WhenAny(breakfastTasks);

breakfastTasks.Remove(finishedTask);

}

Juice oj = PourOJ();

}

Note that everytime await is used, the method being invoked must be asynchronous. If we don’t need to await the result, but just want to run a method asynchronously, we can return void instead of returning Task<T>. The convention for asynchronous methods is to append them with ‘Async’. For asynchronous methods that return void (don’t need to be awaited), prepend them with either Begin or Start.

TODO: Fix --> Database Incorporation:

In order to use EF in our projects, there are a few packages that we need to install. On Windows, this would normally be done through the NuGet package manager. Confusingly, on Linux there is a subcommand in the dotnet cli called nuget, however, this is not actually the command that we want to use. In order to add packages on Linux, we must use the “add package” subcommand. Here are a list of every package you’ll want/need:

dotnet add package Microsoft.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Sqlite

dotnet add package Microsoft.EntityFrameworkCore.Tools

dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design

dotnet add package Microsoft.EntityFrameworkCore.Design

The first package installs EF itself. The second package installs Sqlite support for EF if we want to use Sqlite databases. The third package allows us to use scaffolding which will be very useful.

On Linux, we’ll need to install an additional tool to be able to scaffold from the cli. To do this, run *dotnet tool install --global dotnet-aspnet-codegenerator.*

To scaffold using our dotnet-ef tool from the cli, we can run *dotnet aspnet-codegenerator controller -name <ControllerName> -async -api -m <ModelName> -dc <ContextName> -outDir Controllers.*

Open your <project\_name>.csproj file and delete the line <Nullable>enable</Nullable>

Static Content:

As of web 2.0, we began to drift away from serving static content (a.k.a. assets), and opted to start serving data driven content (i.e. dyanamic content, such as Razor pages, for example). Assets would be anything located in the wwwroot directory in your .NET project. Standard content that goes here are JavaScript files, images, CSS files, and HTML files. The wwwroot directory is called the “web root”, as it is where our web pages search by default for the assets being served. The web root directory can be modified by using the WebApplicationOptions parameter in the CreateBuilder() function in Program.cs. For example:

var builder = WebApplication.CreateBuilder(new WebApplicationOptions {

Args = args,

// Look for static files in webroot

WebRootPath = “webroot”

});

In order to use static files in our project, we require the UseStaticFiles() middleware, which can be added with app.UseStaticFiles(); in Program.cs. Once this is enabled, we can reference the web root directory in our HTML pages with the tilde (~). For example:

*<img src=”~/images/MyImage.jpg” class=”img” alt=”My image” />*

Note that the tilde replaces “webroot”. The src URI is equivallent to https://<host>/images/MyImage.jpg.

Static files can also be served from a decendant directory of the web root directory. To do this, we can add the StaticFileOptions parameter to the UseStaticFiles middleware. For example:

app.UseStaticFiles(new StaticFileOptions {

FileProvider = new PhysicalFileProvider(

Path.Combine(env.ContentRootPath, “MyStaticFiles”)

),

RequestPath = “/StaticFiles”

});

Here, we setup the physical file provider with a path to the decendant directory that we want to use for storing our assets. Assuming that web root is a directory called webroot in this example, then env.ContentRootPath will evaluate to webroot and therefore, our resulting path will be webroot/MyStaticFiles. Next, we set the RequestPath to be /StaticFiles. The RequestPath is what the user actually needs to request in the URL to retrieve the asset. In other words, it maps to webroot/MyStaticFiles in our local project structure. We could serve assets like so:

*<img src=”~/StaticFiles/images/red-rose.jpg” class=”img” alt=”A red rose” />*

HTML Static Files:

You may have noticed that many popular applications use static web pages as their main page when loading the application. Typically this will be a generic web form for entering login credentials (think of Amazon, Ebay, Gmail, Facebook). This is because static files can be cached, and therefore, we can reduce server load if we first serve the cached content immediately and begin loading the dynamic content in the meanwhile. ASP.NET can cache files for us, also by using the StaticFileOptions parameter in the UseStaticFiles() middleware. Example:

var cacheMaxAge = 604800; // 7 days (in seconds)

app.UseStaticFiles(new StaticFileOptions {

OnPrepareResponse = ctx => {

ctx.Context.Response.Headers.Append(

“Cache-Control”, $”public, max-age={cacheMaxAge}”);

);

}

});

Not only can we cache assets, we can also serve them as default documents or landing pages. In order to do this, we use the UseDefaultFiles() middleware. The UseDefaultFiles() middleware must be served before the UseStaticFiles() middleware. The UseDefaultFiles() middleware searches for a couple of files in wwwroot and picks the first one from the following ordered list:

- default.htm

- default.html

- index.htm

- index.html

This can be overriden by passing a DefaultFilesOptions instance as a parameter:

var options = new DefaultFilesOptions();

options.DefaultFileNames.Clear();

options.DefaultFileNames.Add(“mydefault.html”);

app.UseDefaultFiles(options);

app.UseStaticFiles();

Directory Browsing:

Think of a page such as you would see on an FTP server, where we are able to see a directory tree of files beginning at some root directory and then download files or read them in the browser. ASP.NET allows us to do this, however, it is disabled by default for security reasons. We can enable it by adding the AddDirectoryBrowser() service, followed by the UseDirectoryBrowser() middleware:

...

services.AddDirectoryBrowser();

...

app.UseDirectoryBrowser(new DirectoryBrowserOptions {

FileProvider = new PhysicalFileProvider(

Path.Combine(env.WebRootPath, “images”)

),

RequestPath = “/MyImages”

});

UseFileServer:

UseFileServer combines the functionality of UseStaticFiles(), UseDefaultFiles(), and UseDirectoryBrowser() (optional). All we have to do to use the file server is enable the UseFileServer() middleware. In order to have it use UseDirectoryBrowser(), we set the enableDirectoryBrowsing parameter to true e.g. *app.UseFileServer(enableDirectoryBrowsing: true);*. By adding the UseFileServer() middleware, we do not need to add UseStaticFiles() and UseDefaultFiles(), since UseFileServer() will add those for us.

Asset Considerations:

Be warned that for cross-platform compatibility, we need to be aware of how systems handle case sensitivity for content that is exposed via UseDirectoryBrowser() and UseStaticFiles(). Windows does not require case sensitivity for files, however, Mac and Linux do, so primarily programmers on Windows ought to be mindful of this. If using Apache IIS for hosting, note that ASP.NET Core applications use the ASP.NET Core Module to forward all requests, including the service of static files, therefore, the IIS static file handler will not be used. Static content should always be located withing the web root directory and separated from the .cs files and Razor pages. This is to reduce the risk of server-side code being leaked. Speaking of leaking, UseDirectoryBrowser and UseStaticFiles can leak secrets. Disabling directory browsing in production is highly recommended.

A Review of Model State:

Previously, I suggested that the Model Binder fails silently in most circumstances and that we must check ModelState.IsValid to know whether or not binding succeeded. There are various ways of notifying the user if model binding failed, however. The most obvious solution is to return an error page view. For example:

public async Task<IActionResult> SomeAction()

{

if (!ModelState.IsValid)

{

return View(“Error”);

}

...

}

If we would like to re-bind a model to update its state we can use the TryValidateModel() function like so:

public async Task<IActionResult> SomeAction()

{

var modifiedReleaseDate = DateTime.Now.Date;

Movie.ReleaseDate = modifiedReleaseDate;

ModelState.ClearValidationState(nameof(Movie));

if (!TryValidateModel(Movie, nameof(Movie)))

{

return View(“Error”);

}

...

}

ASP.NET provides the programmer with validation attributes, which are attributes that will validate fields in a form on the server-side. Some examples of built-in validation attributes are [CreditCard], [Compare], [EmailAddress], [Phone], [Range], [RegularExpression], [Required], [StringLength], [Url], [Remote], etc. These attributes should decorate the appropriate properties in our model class. Most, if not all, of these attributes allow you to override the default error message by setting the ErrorMessage parameter like so:

*[StringLength(8, ErrorMessage = “{0} length must be between {2} and {1}.”, MinimumLength = 6)]*

*public string Name { get; set; }*

It is the duty of the programmer to update/refresh the view when binding fails due to one of these validation attributes failing so that the error message is displayed.

Alternatively, or in conjunction with validation attributes, we can have client-side validation which happens in the form itself. The submit button will run some JavaScript function that will either submit the form or display the error message that we supply. Note, however, that in order to use JS we require the \_ValidationScriptsPartial.cshtml file in our Views/Shared directory. This file contains script tags which link the appropriate JS scripts from the web root directory. This file will be added automatically when scaffolding. Additionally, in order to use the functions as partial views in our Razor pages, we need the following somewhere in the .cshtml file:

*@section Scripts {*

*@{ await HTML.RenderPartialAsync(“\_ValidationScriptsPartial”); }*

*}*

This will asynchronously load our JS functions linked by the script tags in \_ValidationScriptsPartial into the current Razor page so that we can render them as partial views when needed. Validation can be added to a form like so:

<div class=”form-group”>

<label asp-for=”FirstName” class=”control-label”></label>

<input asp-for=”FirstName” class=”form-control” />

<span asp-validation-for=”FirstName” class=”text-danger”></span>

</div>

The key point here is the asp-validation-for tag helper, which is set in the span tag.

We’ve looked at validation attributes and server-side validation. A third method of validating a field is to use the [Remote] validation attribute to point to an action that will act as a validation handler (useful for more customized validation i.e. more than just an error message). For example, say that we have a user entering an email, and we are considering the email to be a form of identification (like a username). We can add an action that checks to see if the email is already in use or not, and then return that information to the view. Here is the example:

// In the model class (User.cs)

[Remote(action: “VerifyEmail”, controller: “Users”)]

public string Email { get; set; }

// In the controller (UserController.cs)

[AcceptVerbs(“GET”, “POST”)]

public IActionResult VerifyEmail(string email)

{

if (!\_userService.VerifyEmail(email))

{

return Json($”Email {email} is already in use.”);

}

return Json(true);

}

File Upload Methods:

We looked at serving static content i.e. assets to the user, but what if the user wants to upload a file to the server? This is very common actually (consider any social media platform – we must allow the user to upload images frequently).

There are two primary methods of uploading a file: buffered model binding, and unbuffered streaming. Buffered model binding is for smaller files, whereas unbuffered streaming is for larger files. Buffered model binding binds a file from the Razor page to an IFormFile instance. If the file size is too large, or there are too many requests to buffer uploads, the site may crash! Unbuffered streaming, on the other hand, recieves a multipart request (this is when we split the file into chunks and send each chunk individually). Unbuffered streaming has no improvement in terms of performance, but it does reduce the risks of the site crashing.

Buffered Model Binding:

In order to support file uploads, HTML forms must specify an encoding type. To do this, we use the HTML enctype attribute in our form tag. In our input tag, we ought to set the type attribute to “file”. To support uploading a batch of files all at once, we add the ‘multiple’ attribute to the input tag as well. For example:

<form enctype=”multipart/form-data” method=”post”>

<input asp-for=”FileUpload.FormFile” type=”file” multiple>

<input type=”submit” value=”Upload” />

</form>

Within the controller, we’ll use an IFormFile as a parameter in our action, as mentioned. If the ‘multiple’ attribute was set in the HTML file input tag, then we need to use one of the following: IFormFileCollection, List<IFormFile>, IEnumerable<IFormFile>. For example:

public async Task<IActionResult> UploadFile(IEnumerable<IFormFile> files)

{

...

foreach (var formFile in files)

{

var filePath = Path.Combine(\_config[“StoredFilesPath”], Path.GetRandomFileName());

using (var stream = System.IO.File.Create(filePath))

{

await formFile.CopyToAsync(stream);

}

}

...

}

This action will accept an enumerable list of IFormFiles and then for each file it recieves, it will create a path based upon whatever \_config[“StoredFilesPath”] returns + a random file name. It will create a file stream object and then copy the file to that stream (which basically places the file into the location that we specified with filePath). Because it is possible for attackers to inject malicious code by using scripted file names, it is best that we use GetRandomFileName() or GetTempFileName() to name our files. Note that GetTempFileName() creates an empty file, which is not always desirable.

It is possible to manually validate the file extension of a file being uploaded. Note, however, that this is not a reliable method of checking what type of file it actually is. Linux, for example, does not rely on file extensions, and even on Windows we can rename files to remove their extensions. For basic validation, the following code does the job:

private readonly string[] permittedExtensions = { “.txt”, “.pdf” };

...

var ext = Path.GetExtension(uploadedFileName).ToLowerInvariant();

if (string.IsNullOrEmpty(ext) || !permittedExtensions.Contains(ext))

{

// Handle the invalid extension

}

To improve upon the basic file extension validation, we can perform file signature validation. This will match the provided extension with the header of the file, and determine if they match. Unfortunately, the size of the header in bytes must be known beforehand, which can be unreliable. For example: *var headerBytes = file.ReadBytes(size);* Note that file size validation can also be included easily by simply checking *if (file.Length > \_fileSizeLimit) { ... }*.

Data Storage Types:

There are several places in which we can store uploaded user content. Most obvious would be a database. DBs are better for small files. They are the fastest method for storing files. They also have the benefit of allowing the user to return them with other user-related data. Finally, they are less expensive than something like cloud storage. The second method of storing uploaded content is on the web server itself. This is good for large file uploads. The application requires R/W access to the web server’s storage, which comes with a security risk. It is also less expensive than cloud storage. Finally, there is cloud storage, which has better scalability, reliability, and security over both the DB and web server options.

As we used Azure in the course, I will discuss which options Azure provides for cloud storage. Azure’s storage solution is actually called “Azure Storage”. It provides methods for storing images, transactions (e.g. tweets and facebook posts), and message queues (email blasts, texting, etc.). To handle all of these types of data, Azure Storage provides the following storage types:

**Blob Storage:** BLOB stands for Binary Large Objects and is a common format for large unstructured data such as images, video/audio files, or executable binaries. The maximum file size for a BLOB on Azure Storage is 195GB.

**Table Storage:** Table storage uses NoSQL’s Key:Value pair system. It is semi-structured and uses JSON to serialize data. Using table storage is a cheaper alternative to paying for an entire database in the cloud.

**Queue Storage:** Queue storage is used for message queues. It is good for when you need to decouple the application, and also if you require asynchronous messaging.

**File Storage:** File storage is for on-premise use. It provides a distributed network file share (once again, think of an FTP server). Employees at a company could access important resources through Azure Storage’s file storage.

**Disk Storage:** Disk storage provides block-level storage volumes for Azure VMs (also for on-premise use). This is good for when you require restricted access to the VM that the disk is attached to.

BLOB Storage:

On Azure Storage, BLOB storage are setup through a specific structure. At the top level, we have storage accounts, which are a unique namespace in Azure for your data. Every object that you store in Azure Storage has an address that includes you unique account name. Each account may have containers. A container essentially nothing more than a glorified folder where we keep our blobs. For example, a container for image blobs, or a container for video blobs. Note that there are no limits on how many containers an account can have, and there are no limits on how many blobs a container can store. Finally, we have the blob data itself. Blob data comes in three types: block blobs, append blobs, and page blobs.

In order to use Azure Storage, we require the Azure Storage libraries, which are not part of .NET Core. The libraries can be installed as a single NuGet package called Azure.Storage.Blobs. This provides the following structures for us to use:

**BlobServiceClient:** Manipulates Azure Storage resources and blob containers. This is the top-level namespace for the blob service. It requires its own connection string at initialization e.g. *new BlobServiceClient(blobConnection);*where blobConnection is the connection string that is also included in appsettings.json and appsettings.Development.json.

**BlobContainerClient:** Manipulates Azure Storage containers and the blobs that they contain.

**BlobClient:** Manipulates Azure Storage blobs.

**BlobDownloadInfo:** Represents the properties and content returned from downloading a blob.

Reducing Security Risks:

It is advised that you adhere to the following suggestions to mitigate security risks when using Azure Blob Storage:

- Upload files to a dedicated file upload area, preferably a non-system drive

- Do not persist uploaded files in the same directory tree as the app

- Use a safe file name determined by the app. Don’t use a file name provided by the user or the untrusted file name of the uploaded file

- Allow only approved file extensions for the app’s design specification

- Verify that client-side checks are performed also on the server-side, as client-side checks are easily circumvented

- Check the size of an uploaded file

- When files shouldn’t be overwritten by an uploaded file with the same name, check the file name against the database or physical storage before uploading

- Run a virus/malware scanner on uploaded content before it is stored

Making HTTP Requests:

To make an HTTP request in ASP.NET, we use an HttpClient object. An IHttpClientFactory can be registered and used to configure and create HttpClient instances in an app. There are several ways that we can use an IHttpClientFactory. These methods can be broadly summarized as: the basic usage, named clients, typed clients, and generated clients.

The basic usage is good for simple use-cases or prototyping. The IHttpClientFactory is registered by calling AddHttpClient(). Here is some example code:

public class Program

{

public void Main()

{

...

services.AddHttpClient();

...

}

}

public class BasicUsageModel

{

IHttpClientFactory \_clientFactory;

public BasicUsageModel(IHttpClientFactory clientFactory)

{

\_clientFactory = clientFactory;

}

public async Task OnGet()

{

var request = new HttpRequestMessage(HttpMethod.Get,

“<https://api.github.com/repos/dotnet/AspNetCore.Docs/branches>”);

request.Headers.Add(“Accept”, “application/vnd.github.v3+json”);

request.Headers.Add(“User-Agent”, “HttpClientFactory-Sample”);

var client = \_clientFactory.CreateClient();

var response = await client.SendAsync(request);

if (response.IsSuccessStatusCode)

{

using var responseStream = await response.Content.ReadAsStreamAsync();

Branches = await JsonSerializer.DeserializeAsync<IEnumerable<GitHubBranch>>(responseStream);

}

else

{

GetBranchesError = true;

Branches = Array.Empty<GitHubBranch>();

}

}

}

Named clients are a good choice when the app requires many disctinct uses of the HttpClient, and/or when many HttpClients have different configurations. Rather than setup the request headers in our OnGet() function, we can include that information within the AddHttpClient() service in Program.cs like so:

services.AddHttpClient(“github”, c => {

c.BaseAddress = new Uri(“<https://api.github.com/>”);

// Github API versioning

c.DefaultRequestHeaders.Add(“Accept”, “application/vnd.github.v3+json)”;

// Github requires a user-agent

c.DefaultRequestHeaders.Add(“User-Agent”, “HttpClientFactory-Sample”);

});