Modern software development with `C#` and `ASP.NET`

Caybo Kotzé

A little prelude into the world of .NET from the basics to building something useful.

1 Interlude

1.0.1 About me

Currently, I work at Capgemini as a software engineer. C# with ASP.NET and JavaScript is my primary focus, however I have also worked with Java and dabbled with Python, Rust and GoLang. I have written a few C# libraries to make life easier for some developers. The ones I use most frequently in my own projects is the sql-migration runner, Dapper.CQRS (to use Dapper with the CQRS pattern) and SqlQueryBuilder (A fluent library to build up SQL queries). The ones You can check out my public GitHub profile to see the types of things I work on and am interested in. https://github.com/caybokotze

1.0.2 Booklet Resources

All the exercises references in this booklet can be found at this link: https://github.com/caybokotze/modern-software-development-with-dotnet. The resources are recommended to follow along with the exercises later on in this booklet.

2 Introduction to ASP.NET

ASP.NET (Active Server Pages) is a brilliant framework developed by Microsoft which allows us as developers to leverage C# to write a large variety of software very quickly and easily. The ASP.NET framework is primarily a WEB API framework which we can use to build API's using C#. Although there are other vendors which provide frameworks as an alternative to ASP.NET also written in C#, ASP.NET is the best known and best supported framework. The typical ASP.NET project consists of a MVC (Model, View, Controller) pattern. This is not required but generally good practice for web based projects. There are two ways to make use of the modern ASP.NET framework, the first is by creating controllers following the MVC pattern and the other is something newer called minimal API's. Minimal API's resemble a architectural syntax closer to what you will come across when building API's using python flask or node.js express.

C# was created in 2000 by Anders Hejlsberg and released in 2002. C# is a *strongly typed*, *statically typed* and *managed language*. This means variable types can not be changed once defined and the language implementation is managed by the Runtime. The runtime has a garbage collector which removes variables which are out of the execution scope to avoid memory leaks. Although writing unmanaged code is also possible.

C# was created to tackle the emerging market for web development at the time as a direct competitor to Java which was doing the same thing. Doing web development at the time meant using Visual Basic or C++ which doesn't really suit web development very well. So C# was created as a high performance alternative to those languages.

Java was aiming to be the C and C++ killer in the 1990's and so Microsoft put together a team to build an alternative of their own. Some people claim that Microsoft copied Java which is not really true when looking at the history between Microsoft and Sun Technologies (Acquired by Oracle). Microsoft was making use of Java and modified the language to suit their needs into a language which was referred to as J++. Sun then sued Microsoft for doing this, so that forced them to create a language objectively better than Java in many ways. This is easy to see by all the features available in C# that are not available in Java out of the box.
Download a copy of the ASP.NET framework by visiting https://asp.net

2.1 The ASP.NET SDK

The SDK contains all of the tools you require as a developer, including the *dotnet-cli*, the *roslyn compiler*, the *kestrel web server* as well as the *dotnet runtime (CLR)*. Each of these tools are responsible for a slightly different part of the what makes a ASP.NET application work.

2.1.1 ASP.NET CLR

ASP.NET is a managed language, which means that it doesn't get directly compiled to machine code, but rather an intermediary language called *MSIL (Microsoft Intermediary Language)*. The ASP.NET CLR is responsible for executing this IL code on the host machine. For ASP.NET Core that means it can run on any device, whether it be a x86 or x64 host device. The CLR is primarily only responsible for running IL compiled code, not compiling it. That is the job of the compiler.

While Java has the JVM C# has the .NET CLR and now the .NET Core CLR. ASP.NET Core version 1 was released in June 27th 2016. Like the JVM this new CLR runtime can host C# binaries on any device. This includes Windows, Mac, Linux and Android devices.

2.1.2 The role of the CTS

The C# Common Type System is responsible for defining the valid set of types which are interpreted by the CLR. This is responsible for defining the rules that govern what is possible with C# and what isn't valid code.

2.1.3 The Roslyn Compiler

The compiler responsible for compiling C# code into *IL* is the *Roslyn* compiler. This compiler is automatically bundled with the *ASP.NET SDK*.

2.1.4 Dotnet CLI

The dotnet command line interface allows you to do the following.

- Build C# apps
- Run C# apps.
- Run Unit Tests.
- Install/Manage NuGet packages
- Package NuGet packages

There are some additional tools available via the *dotnet-cli tools* command which we can use to de-compile C# applications as well make use of inspection tools like *ildasm*. With this tool we can take an existing compiled MSIL (.dll) and inspect that code and even convert it back to idiomatic C# code. It is technically possible to take a compiled F# sharp application or VB.NET application and convert that to C# code. This is possible because all .NET languages compile to MSIL and can be decompiled back into idiomatic code.

3 ASP.NET Framework vs ASP.NET Core

3.1 The evolution of .NET

Something important to understand are the changes that Microsoft have undergone within the ASP.NET landscape, with the shift from ASP.NET Framework to ASP.NET Core.

The main difference between .NET Framework and .NET Core is the CLR which the framework is built on-top of. The .NET Framework catered to Web and Desktop application development targeted against Windows x86 and x64. The .NET Core CLR however, can run on Linux, Windows or Mac.

Microsoft's plan is to retire .NET Framework as a platform eventually as more of the company resources are going into the development of ASP.NET Core.

ASP.NET Framework was introduced in January 2002 and has seen its fair share of changes since the launch of .NET Core in 2016. The introduction of .NET Core in 2016 signified a shift in focus as Microsoft moves towards more of a unified .NET platform. We saw this in 2020 when Microsoft released .NET 5 which combined a lot of the functionalities of .NET Framework and what was then .NET Core 3. This makes a lot of sense as opposed to maintaining 2 different .NET types. ASP.NET Framework does still exist and is still supported but a lot more effort is being spent on improving a unified version ASP.NET.

The current version of .NET Core (.NET 6 at the time of writing) has seen massive performance improvements over .NET Framework with some functionality performing up to 400% faster. In fact there are aspects of C# that now run very close to as fast as the same functionality written in C++ which is a very impressive feat to achieve. When compared to Java and Python the language is generally much faster. Some types of operations are 2X or more faster that the same code written in Java and 10 to 100 000 times faster than python. No that is not a mistake, python is a great language to learn quickly but because of how the language was written isn't very good performance-wise.

3.2 ASP.NET Framework

ASP.NET Framework, the predecessor of *ASP.NET Core* is the framework that Microsoft created back in 2002. The .NET Framework is currently still the go to for Windows UWP WinForm applications.

3.3 ASP.NET Core

The initial release of *ASP.NET Core 1.0* was in July 2016. As of the time of writing the most recent major ASP.NET version is ASP.NET Core 6. At the end of the year (2022), we should have our first stable release of *ASP.NET 7* which brings about a lot of new features not yet in the ASP.NET Core 6 framework.

3.4 Comparing C# code to Java code

C# is a *JIT* based managed statically typed language created by Microsoft. Java is a *AOT* based managed statically typed language created by Sun Technologies, which was later acquired by Oracle.

I have personally worked with C# as well as Java and definitely have a strong preference syntactically in favor of C# and the power and simplicity that it provides. In this example outlined below you can see a snippet of the two different languages which illustrates how they are similar but also different.

3.4.1 Simple Pojo/Poco's

Note in the example below the differences between Java and C#. Java does not have a set implementation for a "property" natively. Instead it has accessor and mutator methods. In C# the equivalent of this is called a *getter* and *setter*. Another difference is that Java methods and variables are commonly camelCase. In C# the convention follows ot create methods as PascalCase instead.

A POJO is a plain old Java object. A POCO is a plain old C# object. They represent the most basic types of classes within Java and C# respectively, which often to not contain any logic inside. They are purely used to map out how data is structured within an application.

Java User class POJO

```
public class User {
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public void setFirstName(String name) {
        this.firstName = name;
    }

    public String getLastName() {
        return this.lastName;
    }

    public void setLastName(String name) {
        this.lastName = name;
    }
}
```

Properties in C# with getters and setters to the same job as accessors and mutators, but more succinctly. There are libraries like *lombok* in Java which can be used to achieve the same look and feel as C# properties, but to me that feels like a bit of a hack. C# properties are first-class citizens and is built into the language. You will notice in the example below that to achieve the same result does require a lot more boilerplate code in Java than what is required in C#. And in every application you will most likely have dozens to hundreds of POJO / POCO's, so that work you don't have to do can add up.

C# User class POCO

```
public class User {
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

3.4.2 Find the first item in a list

Just to rub it in a little bit, here is a typical example of something commonly done which clearly shows some of the differences between C# and Java.

Java

```
class Stop {
    private final String stationName;
    private final int passengerCount;

    // Stop class constructor
    Stop(final String stationName, final int passengerCount) {
        this.stationName    = stationName;
        this.passengerCount = passengerCount;
    }

    public void setStationName(String stationName) {
        this.stationName = stationName;
    }

    public void setPassengerCount(int passengerCount) {
        this.passengerCount = passengerCount;
    }

    public String getStationName() {
        return this.stationName;
    }

    public int getPassengerCount() {
        return this.passengerCount;
    }
}

public static void main(String[] args) {
    List<Stop> stops = new LinkedList<Stop>();
    stops.add(new Stop("Station1", 250));
    stops.add(new Stop("Station2", 275));
    stops.add(new Stop("Station3", 390));
    stops.add(new Stop("Station2", 210));
    stops.add(new Stop("Station1", 190));

    Stop firstStop = stops.stream()
            .filter(e -> e.stationName.equals("Station3"))
            .findFirst()
            .orElse(null);

    System.out.printf("At the first stop at Station1 there were %d passengers in the train.",
        firstStopAtStation1.passengerCount);
}
```

Note in this example below the *var* keyword. This is something that was introduced into C# in 2007 and allows a developer to *implicitly* define the type of variables. The concrete type is then resolved at compile time and results in zero overhead at run time. The behaviour is still strongly-typed, so once a variable type is declared it cannot be changed, unlike in languages like Python or JavaScript.

C#

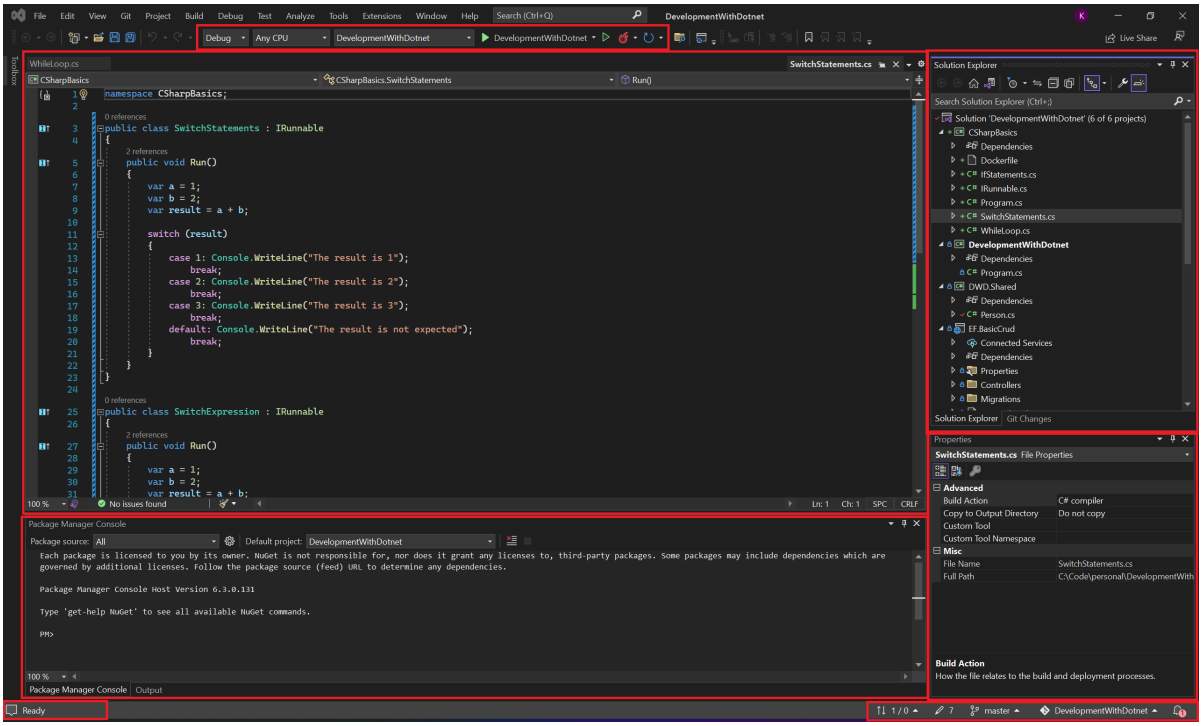
```
class Stop {
    public string StationName { get; set; }
    public int PassengerCount { get; set; }
}

public static void main(String[] args) {
    var stops = new List<Stop>() {
        new("Station 1", 250),
        new("Station 2", 275),
        new("Station 3", 390),
        new("Station 2", 210),
        new("Station 1", 190)
    };

    var firstStop = stops.First(f => f.StationName == "Station 3");
    Console.WriteLine($"At the first stop at station 1 " +
    "there were {firstStop.PassengerCount} passengers in the train");
}
```

4 Getting started with Visual Studio

I'll keep this section brief. C# is *IDE* agnostic, meaning you can use notepad if you want to to write C# code. However, the most popular *IDE's* are *Visual Studio*, *Visual Studio Code* and *Jetbrains Rider*. Below is a screenshot of Visual Studio



The screenshot above outlines the most important parts of visual studio to familiarize with when using *Visual Studio* as your development environment of choice. The biggest block in the middle is your solution window, where the work happens.

On the right of the solution window is the solution explorer which you can use to navigate files, create files and delete files. Below that is the property window which you can use to view the metadata of the selected file in the solution explorer. Below the solution window is the package manager console which is useful when working with the *dotnet cli* or any commands you need to run in a terminal.

Above the solution window are the options we have to run and debug our application. This will be useful when debugging applications and being able to step through break points. Below the package manager console on the bottom left of the screen we have our indexer summary, which reflects whether the project has been fully indexed or not.

On the bottom right hand-side we have our *git* source control overview. We can use this to switch between git branches, make commits and view git commit history.

5 Introduction to C#

This section will review some of the basic language features that C# has to offer. This will not be an extensive covering of all the C# features. C# is a relatively large language in terms of keywords and complexity, so I would recommend a book called Head First C# by Andrew Stillman and Jennifer Greene to familiarize yourself with all the C# language features.

5.1 Prerequisites for application logic and string literals

5.1.1 Equality and Relational Operators

Operator	Example	Explanation
==	if(age == 30)	Returns true only if each expression is the same
||	if(name != "John")	Returns true if each expression is different
<	if(age < 18)	Returns true if age is less than 18
>	if(age > 18)	Returns true if age is greater than 18
<=	if(age ≤ 18)	Returns true if age is less than or 18
>=	if(age ≥ 18)	Returns true if age is greater than or 18

5.1.2 Logical Operators

Operator	Example	Explanation
&&	if(age == 40 && name == "John")	AND operator. Returns true if all operations are true
||	if(age == 22 || name == "Bob")	OR operator. Returns true if either operation is true
!	if(!myBoolean)	NOT operator. Returns true if false, or false if true

5.1.3 Escape Characters

Escape character	Explanation
\'	Inserts a single quote into a string literal
\"	Inserts a double quote into a string literal
\\	Inserts a backslash into a string literal
\a	Triggers a system alert (normally a audio cue)
\n	Inserts a new line (on Windows)
\r	Inserts a carriage return
\t	Inserts a horizontal tab into the string literal

5.2 Conditional Statements

Conditional statements lie at the heart of every programming language. It is how we use classes, methods and variables to create a flow of logic to solve problems. Ultimately there are several ways in which we can express these conditions, however these ones are the most important to be aware of.

5.2.1 If statements

A *if* statement is the most basic type of conditional statement. We can use logical and comparative operators to write very simple to more complex if statements. Here are some examples.

```
var a = 1;
var b = 2;

if (a == b)
{
    Console.WriteLine("a is equal to b");
}

if ((a == 1 || b == 1) && a + b >= b) {
    Console.WriteLine("a or b is 1 and the sum of a and b " +
    "is greater than or equal to the value of b");
}
```

5.2.2 Switch Statements

A switch statement can be very useful to write multiple if statements more succinctly. If you have more than 2 if statements written against the same condition, then making use of a switch statement is the obvious choice.

```
var a = 1;
var b = 2;
var result = a + b;

switch (result)
{
    case 1: Console.WriteLine("The result is 1");
        break;
    case 2: Console.WriteLine("The result is 2");
        break;
    case 3: Console.WriteLine("The result is 3");
        break;
    default: Console.WriteLine("The result is not expected");
        break;
}
```

5.2.3 Switch Expressions

Switch expressions can only be used when the result of the expected matched value is not void. However, for use cases where there is opportunity to make use of that feature, it does read in a easy to understand, more succinct way than a traditional switch/case statement. The _ (underscore throwaway) is used in place of the *default* code block in a switch/case statement.

```
var a = 1;
var b = 2;
var result = a + b;
var resultPlusOne = result switch
{
    1 => 1 + 1,
    2 => 2 + 1,
    3 => 3 + 1,
    _ => 0
};
```

5.3 Loops

When solving problems where any type of iteration is required over or across a data set we use loops. Depending on what type of iteration needs to happen we can employ different types of loops that best fit the use-case. Generally speaking if the length of the data-set is not predetermined or known then we would make use of a *while loop* whereas if the data length is known we can make use of a *for* or *foreach* loop.

5.3.1 While Loops

In this example below an infinite loop is created where the condition of the while loop will always be true. The only way to break out of the infinite loop is with the *break* keyword. This is a good example of a loop that can continue infinitely until a specific condition is met. It needs to be kept in mind that an infinite loop can easily be created whether *intentionally* or *not*. So use while loops carefully ;)

```
var i = 1;

while (true)
{
    Console.WriteLine($"This is iteration number {i}");

    if (i >= 10)
    {
        break;
    }

    i += 1;
}
```

5.3.2 For Loops

A for loop is useful when the data set is predetermined. This is the most common type of loop encountered.

```
for (var i = 0; i <= 10; i++)
{
    Console.WriteLine("This is iteration number {i}");
}
```

5.3.3 ForEach Loop

A for loop is a wrapper around a for loop, so we don't need to define a variable and define the length of the array being iterated over, making them easier to use.

```
var people = new List<Person>()
{
    new()
    {
        FirstName = Faker.Name.First(),
        LastName = Faker.Name.Last(),
        Age = DateTime.UtcNow.Year - Faker.Identification.DateOfBirth().Year
    }
};

foreach (var person in people)
{
    Console.WriteLine(JsonSerializer.Serialize(person));
}
```

5.3.4 Recursive Loops

A recursive loop can be achieved by creating a method which calls itself. This type of loop can be used in more or less the same way as a while loop. A recursive loop is useful for object parsing where the depth of a nested data structure is not defined beforehand.

```
private int i = 0;

public void ExecuteLogic()
{
    i += 1;
    Console.WriteLine("The value of i is {0}", i);
    if (i < 10)
    {
        ExecuteLogic();
    }
}
```

5.4 Exception Handling

Exceptions are a useful way of breaking the implementation of code if something unexpected occurs that can not be handled further. This will completely stop the execution of the code unless that exception is wrapped with a *try catch* statement. Then we can decide to re-throw the exception or suppress it to allow our code to continue executing. This is a very useful mechanism and something used extensively within the standard libraries within .NET.

```
public void Run()
{
    try
    {
        DoSomeWork();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}

public void DoSomeWork()
{
    throw new Exception("Something isn't quite right");
}
```

5.5 LINQ

Language-Integrated Query (LINQ) is a powerful tool based on the integration of query capabilities directly into the C# language. LINQ Queries is a first-class language construct in C#, just like classes, methods or delegates. LINQ provides a consistent query experience to query objects (LINQ to Objects), relational databases (LINQ to SQL), and XML (LINQ to XML). Below is an example of what that looks like.

```
var people = new List<Person>()
{
    new()
    {
        FirstName = Faker.Name.First(),
        LastName = Faker.Name.Last(),
        Age = DateTime.UtcNow.Year - Faker.Identification.DateOfBirth().Year
    },
    new()
    {
        FirstName = Faker.Name.First(),
        LastName = Faker.Name.Last(),
        Age = DateTime.UtcNow.Year - Faker.Identification.DateOfBirth().Year
    },
    new()
    {
        FirstName = Faker.Name.First(),
        LastName = Faker.Name.Last(),
        Age = DateTime.UtcNow.Year - Faker.Identification.DateOfBirth().Year
    }
};

// find the first person with the lowest age.
var youngestPersonQuerySyntax = (from person in people orderby person.Age select person)
    .First();
var youngestPersonExpression = people.OrderBy(a => a.Age).First();
var youngestPersonShortExpression = people.MinBy(m => m.Age);

Console.WriteLine("Youngest Person via query: {0}",
    JsonSerializer.Serialize(youngestPersonQuerySyntax));
Console.WriteLine("Youngest Person via expression: {0}",
    JsonSerializer.Serialize(youngestPersonExpression));
Console.WriteLine("Youngest Person via short expression: {0}",
    JsonSerializer.Serialize(youngestPersonShortExpression));
Console.WriteLine("All ages: {0}", JsonSerializer.Serialize(people.Select(s => s.Age)));
```

6 Advanced C#

Here are some fun things you can do with C# that are a little bit more advanced and not always recommended. But it is good to know that C# does offer these powerful features to build anything you can imagine.

6.1 Unsafe Code

Unsafe code is a feature that C# provides you with to directly manipulate variable pointers and addresses in memory. This is not recommended to make use of unless you have a very good reason. It is very easy to make mistakes and the .NET runtime has no control over the code written when being executed in unsafe context.

```
public void Run()
{
    int i = 10, j = 20;
    Console.WriteLine("Safe swap");
    Console.WriteLine("values before safe swap: i = {0}, j = {1}", i, j);
    SafeSwap(ref i, ref j);
    Console.WriteLine("Values after safe swap: i = {0}, j = {1}", i, j);

    Console.WriteLine("Unsafe swap");
    Console.WriteLine("Values before unsafe swap: i = {0}, j = {1}", i, j);
    unsafe { UnsafeSwap(&i, &j); }
    Console.WriteLine("Values after unsafe swap: i = {0}, j = {1}", i, j);
}

public unsafe static void UnsafeSwap(int* i, int* j)
{
    var temp = *i;
    *i = *j;
    *j = temp;
}

public static void SafeSwap(ref int i, ref int j)
{
    int temp = i;
    i = j;
    j = temp;
}
```

6.2 Reflection

6.2.1 What is reflection?

Reflection provides a mechanism to discover the type/Type of objects, assemblies and modules. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties

6.2.2 Reflecting over class properties

In this example we are making use of reflection to print out all the names and values of properties within a initialized class (object). We will be using the *Person* class as an example.

```
var person = new Person()
{
    FirstName = Faker.Name.First(),
    LastName = Faker.Name.Last(),
    Age = DateTime.UtcNow.Year - Faker.Identification.DateOfBirth().Year
};

// print out all the property names for person without reflection
Console.WriteLine("Property names without reflection");
Console.WriteLine(nameof(person.Id));
Console.WriteLine(nameof(person.Age));
Console.WriteLine(nameof(person.FirstName));
Console.WriteLine(nameof(person.LastName));

// print out the object property names alone using reflection.
Console.WriteLine("Property names with reflection");
foreach (var property in typeof(Person).GetProperties())
{
    Console.WriteLine(property.Name);
}

// print out the object properties and its corresponding value
Console.WriteLine("Property name and value with reflection");
foreach (var property in typeof(Person).GetProperties())
{
    Console.WriteLine("Property name: {0}. Property value: {1}",
        property.Name, property.GetValue(person));
}
```

7 SOLID principles & OOP

SOLID principles are the design principles that enable us to write good clean code, specifically in relation to object orientation. Robert C. Martin compiled these principles in the 1990s. These principles provide us with ways to move from tightly coupled code to loosely coupled code. SOLID is an acronym of the following.

7.1 (S) Single Responsibility Principle

SRP is defined as follows. "A class should have one and only one reason to change, meaning that a class should have only one job." This means that every class and class member should only be responsible for one thing. As a result changing the implementation of a method should not effect the outcome of other methods in the same class. The encapsulation of a specific variable state should be contained within one method. Classes can contain more than one member, however it should be related to the function of the encapsulating class – which itself should be self-descriptive.

7.1.1 SRP Example - What not to do

This is an example of what is generally not a good idea. The class name does not describe everything happening within the class. We are registering a user as well as testing email validity. Two things which are not related to each other within the context of implementation.

```
public class UserService
{
    private readonly SmtpClient _smtpClient;

    public UserService(SmtpClient smtpClient) {
        _smtpClient = smtpClient;
    }

    public void Register(string email, string password)
    {
        if (!IsValidEmail(email)) {
            throw new ValidationException("Email is not an email");
        }

        if (IsValidEmail(email)) {

            var user = new Person
            {
                Email = email,
                Password = password
            };

            SendEmail(new MailMessage("mysite@nowhere.com", email) {
                Subject="Foo"
            });
        }
    }

    public bool IsValidEmail(string email)
    {
        return email.Contains("@");
    }

    public void SendEmail(MailMessage message)
    {
        _smtpClient.Send(message);
    }
}
```

7.1.2 SRP Example - What to do

This is generally considdered a better approach, since each class is only responsible for what it describes.

```
public class BetterUserService
{
    private readonly EmailService _emailService;

    public BetterUserService(EmailService emailService)
    {
        _emailService = emailService;
    }

    public void Register(string email, string password)
    {
        if (!_emailService.IsValidEmail(email))
        {
            throw new ValidationException("Email is not an email");
        }

        var user = new User
        {
            Email = email,
            Password = password
        }; // do something with the user

        _emailService.SendEmail(new MailMessage("mysite@nowhere.com", email)
            {
                Subject = "Hey user!"
            });
    }
}

public class EmailService
{
    private readonly SmtpClient _smtpClient;

    public EmailService(SmtpClient smtpClient)
    {
        _smtpClient = smtpClient;
    }

    public bool IsValidEmail(string email)
    {
        return email.Contains('@');
    }

    public void SendEmail(MailMessage message)
    {
        _smtpClient.Send(message);
    }
}
```

7.2 (O) Open Closed Principle

OCP is described as the following. "Objects or entities should be open for extension but closed for modification." This means that the class should be extendable without modifying the class itself.

7.2.1 OCP Example - What not to do

This is an example of code that does not follow the OCP principle. For the area calculator to support additional logic, you would need to extend the functionality of the AreaCalculator class (additional if statements, as well as changing the Shape class). A better approach would be to build the logic in a way that every class is responsible for doing it's own calculations internally. You can see in the code below that each class tries to do too much and this leads to code clutter and would become difficult to maintain as we start adding more shapes.

```
public class AreaCalculator {
    private Shape _shape;
    public AreaCalculator(Shape shape) {
        _shape = shape;
    }

    public double CalculateArea() {
        if (_shape.Type == "circle") {
            return Math.PI * (_shape.Diameter/2) * (_shape.Diameter/2);
        }

        if (_shape.Type == "square") {
            return _shape.Height * _shape.Width;
        }

        return 0;
    }
}

public class Shape {
    public string? Type { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }
    public int Diameter { get; set; }
}
```

7.2.2 OCP - What to do

In this example you can see that we make use of an interface to force the definition of internal functionality to each class itself, so the logic of the implementation is encapsulated in that class. Therefore no additional changes are required when more shapes are added to the code-base. All that needs to happen is inheritance from that interface and the accompanying implementation detail. This way each shape is responsible for defining its own behaviour which results in much cleaner code.

```
public interface IShape {
    double CalculateArea();
}

public class Square : IShape {
    public int Width { get; set; }
    public int Height { get; set; }

    public double CalculateArea() {
        return Width * Height;
    }
}

public class Circle : IShape {
    public int Diameter { get; set; }
    public double CalculateArea() {
        return Math.PI * (Diameter / 2) * (Diameter / 2);
    }
}

public class BetterAreaCalculator {

    public double CalculateArea(IShape shape)
    {
        return shape.CalculateArea();
    }
}
```

7.3 (L) Liskov substitution Principle

LSP is defined as the following. "Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T."

7.3.1 LSP - What not to do

You can see in the example below, this concept violates the OCP principal as well as ISP principal. It violates OCP because the SqlFileManager requires new functionality to be added to support additional functionality. In addition to that the classes and parent classes can not be substituted for each other.

```
public class SqlFile
{
    public string LoadText()
    {
        /* Code to read text from sql file */
        return string.Empty;
    }

    public void SaveText()
    {
        /* Code to save text into sql file */
    }
}

public class ReadOnlySqlFile: SqlFile
{
    public string? FilePath { get; set; }
    public string? FileText { get; set; }

    public string? LoadText()
    {
        /* Code to read text from sql file */
        return string.Empty;
    }

    public void SaveText()
    {
        /* Throw an exception when app flow tries to do save. */
        throw new IOException("Can't Save");
    }
}
```

```
public class SqlFileManager
{
    public List<SqlFile>? SqlFiles { get; set; }

    public string GetTextFromFiles()
    {
        var objStrBuilder = new StringBuilder();

        foreach(var objFile in SqlFiles)
        {
            objStrBuilder.Append(objFile.LoadText());
        }

        return objStrBuilder.ToString();
    }
    public void SaveTextIntoFiles()
    {
        foreach(var objFile in SqlFiles)
        {
            // Check whether the current file object is read-only or not.
            // If yes, skip calling it's
            // SaveText() method to skip the exception.

            if (!(objFile is ReadOnlySqlFile))
            {
                objFile.SaveText();
            }
        }
    }
}
```

7.3.2 LSP - What to do

```
public interface IReadableSqlFile
{
    string LoadText();
}

public interface IWritableSqlFile
{
    void SaveText();
}

public class BetterReadOnlySqlFile : IReadableSqlFile
{
    public string? FilePath { get; set; }
    public string? FileText { get; set; }

    public string LoadText()
    {
        /* Code to read text from sql file */
        return string.Empty;
    }
}
```

```
public class BetterSqlFile : IWritableSqlFile, IReadableSqlFile
{
    public string? FilePath { get; set; }
    public string? FileText { get; set; }

    public string LoadText()
    {
        /* Code to read text from sql file */
        return string.Empty;
    }

    public void SaveText()
    {
        /* Code to save text into sql file */
    }
}

public class BetterSqlFileManager
{
    public string GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)
    {
        var objStrBuilder = new StringBuilder();
        foreach (var objFile in aLstReadableFiles)
        {
            objStrBuilder.Append(objFile.LoadText());
        }

        return objStrBuilder.ToString();
    }

    public void SaveTextIntoFiles(List<IWritableSqlFile> writableSqlFiles)
    {
        foreach (var objFile in writableSqlFiles)
        {
            objFile.SaveText();
        }
    }
}
```

7.4 (I) Interface Segregation Principle

The ISP principle is defined as the following. "A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use."

7.4.1 ISP - What not to do

```
public interface IShapeInterface
{
    double CalculateArea();
    double CalculateVolume();
}

public class Square : IShapeInterface
{
    private readonly int _width;

    public Square(int width)
    {
        _width = width;
    }

    public double CalculateArea()
    {
        return _width * _width;
    }

    public double CalculateVolume()
    {
        // i.e. Not possible
        throw new NotImplementedException();
    }
}
```

7.4.2 ISP - What to do

In this example you can clearly see that it is a lot better to have two separate interfaces to define the behaviour for different classes. This way we avoid having methods inside classes that can not be implemented without violating one of the other SOLID principles.

```
public interface I2DShape
{
    double CalculateArea();
}

public interface I3DShape : I2DShape
{
    double CalculateVolume();
}

public class Rectangle : I2DShape
{
    private readonly int _height;
    private readonly int _width;

    public Rectangle(int height, int width)
    {
        _height = height;
        _width = width;
    }

    public double CalculateArea()
    {
        return _height * _width;
    }
}

public class Cube : I3DShape
{
    private readonly int _width;

    public Cube(int width)
    {
        _width = width;
    }

    public double CalculateVolume()
    {
        return _width * _width * _width;
    }

    public double CalculateArea()
    {
        const int amountOfEdges = 6;
        return _width * _width * amountOfEdges;
    }
}
```

7.5 (D) Dependency Inversion Principle

The DIP principle states the following. "High level modules should not depend on low level modules; both should depend on abstractions." In essence this means that when creating classes that have injected dependencies of other classes, we should not inject the class implementation (concrete implementation), but rather a interface which represents the functionality of that class.

7.5.1 DIP - What not to do

In this example the database implementation class *(MySqlConnection)* is injected directly into the *PersonService*. Although this isn't always terrible there is a better way.

```
public class MySqlConnection
{
    public List<object> Read(string query, object parameters)
    {
        return new List<object>();
    }

    public int Write(string statement, object value)
    {
        return 1;
    }
}

public class PersonService
{
    private readonly MySqlConnection _mySqlConnection;

    public PersonService(MySqlConnection mySqlConnection)
    {
        _mySqlConnection = mySqlConnection;
    }

    public void SavePerson(Person person)
    {
        _mySqlConnection.Write("insert into people (age, first_name, last_name) VALUES (@Age, @Fir
    }
}
```

7.5.2 DIP - What to do

In this example, we can see that instead of injecting the implementation of MySqlConnection, we are using a substitute interface which represents any database connection. By doing this we also make our code easier to reuse in other places, since if we wanted to use a different database implementation we do not have to change existing code. It could be as simple as a two line change in the project as opposed to several files.

```
public interface IDatabaseConnection
{
    List<object> Read(string query, object parameters);
    int Write(string statement, object parameters);
}

public class BetterPersonService
{
    private readonly IDatabaseConnection _databaseConnection;

    public BetterPersonService(IDatabaseConnection databaseConnection)
    {
        _databaseConnection = databaseConnection;
    }

    public void SavePerson(Person person)
    {
        _databaseConnection.Write("insert into people (age, first_name, last_name)" +
            "VALUES (@Age, @FirstName, @LastName)", person);
    }
}
```

8 Unit Testing with NUnit

8.1 A summary of NUnit

The main testing libraries available within C# are *NUnit*, *XUnit* and *MSUnit*. The most popular among those two are XUnit and NUnit. Personally, I prefer NUnit, however both XUnit and NUnit do the job of writing unit tests quite well, however I find NUnit more explicit and flexible and therefore prefer making use of it. Along with NUnit you also will need to make use of a mocking library like *Moq* or *NSubstitute* to mock out classes to be able to test more complex implementation logic easier.

8.2 Testing Services

8.2.1 The Service Implementation

Below is an example of how to implement some basic unit tests for a service.

```
public class SumService : ISumService
{
    public int Add(params int[] values)
    {
        return values.Sum();
    }
}

public class MultiplicationService : IMultiplicationService
{
    public double Multiply(int value, double factor)
    {
        return value * factor;
    }
}

public class MathService
{
    private readonly ISumService _sumService;
    private readonly IMultiplicationService _multiplicationService;

    public MathService(
        ISumService sumService,
        IMultiplicationService multiplicationService)
    {
        _sumService = sumService;
        _multiplicationService = multiplicationService;
    }

    public int GetMeanValue(params int[] values)
    {
        var sum = _sumService.Add(values);
        var divideByTwo = _multiplicationService.Multiply(sum, 1.0 / values.Length);
        return (int) divideByTwo;
    }
}
```

8.2.2 The Test Implementation

View the test project in the resources for a more complete example, but below you can view the gist of what would be required to set up a simple unit test implementation.

```
[TestFixture]
public class MathServiceTests
{
    // (Given)
    [TestFixture]
    public class GetMeanValueTests
    {
        // (When)
        [TestFixture]
        public class WhenThreeParametersAreProvided
        {
            [Test]
            public void ShouldReturnAThirdOfThatValue()
            {
                // arrange
                var sumService = new SumService();
                var multiplicationService = new MultiplicationService();
                var sut = CreateMathService(sumService, multiplicationService);
                // act
                var actual = sut.GetMeanValue(2, 4, 3);
                const int expected = 3;
                // assert
                Expectations.Expect(actual).To.Equal(expected);
            }
        }
    }

    public static MathService CreateMathService(
        ISumService? sumService = null,
        IMultiplicationService? multiplicationService = null)
    {
        sumService ??= Substitute.For<ISumService>();
        multiplicationService ??= Substitute.For<IMultiplicationService>();
        return new MathService(sumService, multiplicationService);
    }
}
```

9 Getting started with Web API's

9.1 ASP.NET Web API fundamentals

9.1.1 Dependency Injection

In software engineering, dependency injection is a design pattern in which an object or function receives other objects or functions that it depends on. A form of inversion of control, dependency injection aims to separate the concerns of constructing objects making use of objects. Doing this leads to more maintainable loosely-coupled code.

We have seen many examples of this before in this booklet. But more explicitly, this is what dependency injection will look like. The service (dependency) of a class is injected into the constructor. As a result, any dependency of a class can be resolved by the service container. This also applied to services of services. The service container will resolve dependencies for every dependency registered in the service container.

```
public class PersonService
{
    private readonly DatabaseDependency _databaseDependency;

    public PersonService(DatabaseDependency databaseDependency)
    {
        _databaseDependency = databaseDependency;
    }

    public void SavePerson(Person person)
    {
        _databaseDependency.SaveObject(person);
    }
}

public class DatabaseDependency
{
    public void SaveObject(object value)
    {
        // save logic...
    }
}

// register the code in your container

builder.Services.AddTransient<PersonService, PersonService>();
```

9.1.2 Middleware

In short, middleware is software that's assembled into an app pipeline to handle requests and responses. Each middleware component is responsible for the following:

1. Chooses whether to pass the request to the next component in the pipeline.
2. Can perform work before and after the next component in the pipeline.

9.2 Creating a small Minimal API

9.2.1 Creating a small API GET request

Let's create a minimal API to view the server information with a simple GET request. You can access this in the git resources or create it from scratch.

```
dotnet new webapi
```

The program.cs file should be made to look something like the below:

```
using System.Text.Json;
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/info", () => JsonSerializer.Serialize(new
{
    Environment.UserName,
    Environment.MachineName,
    Environment.ProcessId,
    Environment.Is64BitProcess,
    Environment.OSVersion
}));
app.Run();
```

Believe it or not, the above is a fully functioning web API created with the .NET Web SDK. There is no code hidden from view. In just a few lines of code we can create fast powerful API's which is pretty amazing. You can run this app by typing *dotnet run* in the same directory as the project. This will run the application using the *kestrel web server*. If you would like to create a web API that accepts a POST submission we can easily do that with minimal API's as well as illustrated in the next section.

9.2.2 Creating a minimal API POST endpoint

By adding a *app.MapPost()*, *app.MapPut()*, *app.MapDelete()* or other hook, we can also accept a body request payload, which is automatically de-serialized (By ASP.NET) from a JSON object, into a C# one.

```
public class Person
{
    public string? Name { get; set; }
}
```

```
using System.Text.Json;
using Minimal.API;

public class Program.cs {
    public static void Main(string[] args) {
        var builder = WebApplication.CreateBuilder(args);
        var app = builder.Build();
        app.MapPost("/environment", (Person person) => $"Welcome dear, {person.Name}");
        app.Run();
    }
}
```

Then you can make a request against the API with a payload that looks like this:

```
url: https://localhost:7183/person
content type is text/json or application/json

{
    "name": "Johnny"
}

Then you will get a response that reads "Welcome dear, Johnny"
```

9.2.3 Creating a POST endpoint with Validation

In this example we will be making use of a library called *FluentValidation* to validate incoming models.

Program.cs Implementation

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddValidator<Person, PersonValidator>();
var app = builder.Build();

app.MapPost("/person", (Validated<Person> person) =>
{
    var (isValid, value) = person;

    return isValid
        ? Ok(value)
        : ValidationProblem(person.Errors);
});

app.Run();
```

The Person Class

```
public class Person
{
    [Key]
    public int Id { get; set; }
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public int Age { get; set; }
}
```

The Validator

```
public class PersonValidator : AbstractValidator<Person>
{
    public PersonValidator()
    {
        RuleFor(m => m.FirstName).NotEmpty().MinimumLength(5);
        RuleFor(m => m.LastName).NotEmpty().MinimumLength(5);
    }
}

public class Validated<T>
{
    private ValidationResult Validation { get; }

    private Validated(T value, ValidationResult validation)
    {
        Value = value;
        Validation = validation;
    }

    public T Value { get; }
    public bool IsValid => Validation.IsValid;

    public IDictionary<string, string[]> Errors =>
        Validation
            .Errors
            .GroupBy(x => x.PropertyName)
            .ToDictionary(x => x.Key, x => x.Select(e => e.ErrorMessage).ToArray());

    public void Deconstruct(out bool isValid, out T value)
    {
        isValid = IsValid;
        value = Value;
    }

    // ReSharper disable once UnusedMember.Global
    public static async ValueTask<Validated<T>> BindAsync(HttpContext context, ParameterInfo param
    {
        // only JSON is supported right now, no complex model binding
        var value = await context.Request.ReadFromJsonAsync<T>();
        var validator = context.RequestServices.GetRequiredService<IValidator<T>>();

        if (value is null) {
            throw new ArgumentException(parameter.Name);
        }

        var results = await validator.ValidateAsync(value);

        return new Validated<T>(value, results);
    }
}
```

9.2.4 Creating a POST endpoint with validation and an injected service

Continuing from the example above the example below defined a POST endpoint that accepts a Person as a [FromBody] bound argument as well as a service passed into the event handler. ASP.NET then figures out whether the argument in the event handler maps onto the request body or whether it is a injected service.

```
public class Person
{
    [Key]
    public int Id { get; set; }
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public int Age { get; set; }
}

public interface IPersonService
{
    string Execute(Person person);
}

public class PersonService : IPersonService
{
    public string Execute(Person person)
    {
        if (person.Age == 0)
        {
            person.Age = 32;
        }
        return JsonSerializer.Serialize(person);
    }
}

// Application body
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddValidator<Person, PersonValidator>();
builder.Services.AddTransient<IPersonService, PersonService>();
var app = builder.Build();

app.MapPost("/service",
    (Validated<Person> person, IPersonService executor) => executor.Execute(person.Value));

app.Run();
```

Something interesting to note is that the lambda function "() => {}" will try to figure out what is being injected into the lambda function and then auto-bind it to the correct attribute ([From...]) in the following order.

1. Route values => [**FromRoute**]
2. Query string => [**FromQuery**])
3. Header => [**FromHeader**])
4. Body => [**FromBody**])
5. Services => ([**FromServices**])

You can read more about this on the official Microsoft documentation for Minimal API's here: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis?view=aspnetcore-6..

9.3 Create a small CRUD API with EFCore

In this section we will be moving from Minimal API's to a more traditional MVC based approach by using controllers instead of the *app.MapGet()* and *app.MapPost()* extension methods that minimal API's offer.

9.3.1 What is EF Core?

Entity framework Core is a ORM (Object Relational Mapper) created by Microsoft. This is a NuGet package which you can add to your .NET application to start interacting with a variety of databases. The job of the ORM is to map C# objects onto results given by databases like MySql, SqlServer, PostgreSql and so on. ORM's also provide us with the freedom to not have to worry about writing SQL statements. In ASP.NET our LINQ expressions are converted to a executable query by Entity Framework. There are other ORM's available like Dapper (a micro-ORM) which might be better suited in some situations where more nuanced queries need to be written. Regardless, for our purposes Entity Framework will do the trick.

You can install entity framework by running:
`dotnet add package Microsoft.EntityFrameworkCore`.

We will be working with *SqlLite* instead of *MySql* or *SqlServer*. This keeps our installation process relatively light and is less time consuming to set up.

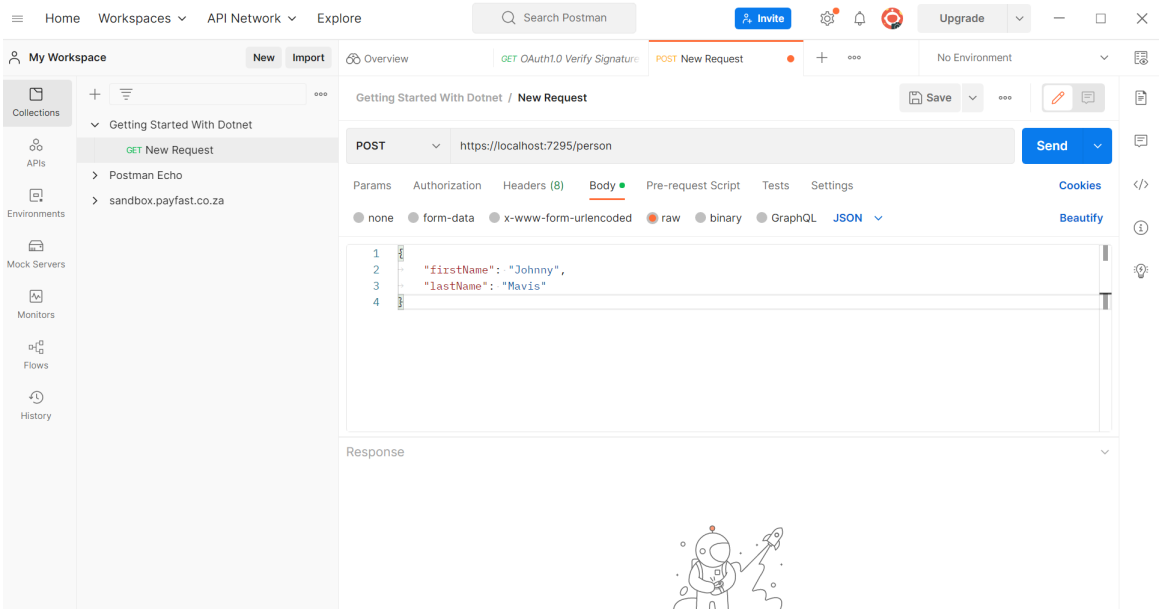
Let's begin by installing SqlLite as well:
dotnet add package Microsoft.EntityFrameworkCore.Sqlite –version 6.0.10

```
This code has been omitted, but can be viewed in the git resources.
```

10 Testing API's with Postman

10.1 What is postman?

Postman is a tool which you can make use of to test API's. You can make use of Postman (or similar tools like Insomnia) to simulate API user interactions that the API consuming client would normally be responsible for making. This can be very useful to test your API and simulate the same requests several times over.



11 Putting knowledge to work

Let's do a little exercise to put the work we have learned into practice by creating a trivia game. The goal would be to create a Server responsible for providing trivia questions as well as persisting the answers. Separate to that we need a front-end UI to allow the user to respond to the question and display the result of the multiple-choice answer which was submitted.

The task at hand will involve creating a three part solution. One API and two Front-end Systems that are responsible for the following.

The first solution will be the API (responsible for generating the trivia question as well persisting responses sent back to the API)

The Second solution will be the front-end to allow the user to post the trivia result to the API. Another responsibility will be for the front-end to display the response from the API notifying the user whether the response was correct or not.

The third solution will be a front-end project to display the submissions which have been made to the server. This will be a simple list view to display the results and periodically update the result set. The list view should also have the option to delete a option within the list.

Which car is the fastest car in the world?

- Bugatti Chiron
- Koenigsegg Jesko Absolut
- Hennesey Venom F5
- Your mother's Toyota Corolla

The solution will be available at https://github.com/caybokotze/modern-software-development-with-dotnet