# General

**What is .Net:**

It’s a managed execution environment to build and run application. It contains the

CLR – Common Language Runtime

FCL – Framework Class Library

**What is C#:**

It’s a strongly-typed, static, object-oriented language that is compiled by the .Net framework into Microsoft Intermediate Language and runs on the .Net or .Net Core CLR

## CLR - Common Language Runtime

Microsoft’s Common Language Runtime – The .Net environment that contains all the services to **run managed code**, including garbage collection, loading libraries etc. The CLR is the **platform abstraction layer** and is different per platform.

## Assemblies

<https://docs.microsoft.com/en-us/dotnet/standard/assembly/>

* Assemblies form the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications.
* Assemblies take the form of executable (.exe) or dynamic link library (.dll) files, and are the building blocks of .NET applications.
* Assemblies are only loaded into memory if they are required. If they aren't used, they aren't loaded. This means that assemblies can be an efficient way to manage resources in larger projects.

## Basic Class Library (BCL)

The basic library of .Net which is part of the CLR standard and is always supported.

## FCL – Framework Class Library

The library of the reusable classes, interfaces and value types like:

* System.Collections
* System.IO
* ASP.Net
* etc

## Managed vs Unmanaged Code

Unmanaged code compiled straight to assembly. Managed code compiles to MIL and managed by the CLR.

## Just in Time Compilation

During compilation of managed code, the code is compiled only to the level of **MSIL** (Microsoft Intermediate Language) or **CIL** which is CPU-independent.

Only on the 1st run of the code, is it compiled to the machine-specific assembly. This MSIL compiled code is called **Assembly** and it contains in addition to the actual code also meta-data on the types the code uses and references to other Assemblies.

## Common Type System (CTS)

The types that the CIL support and can be used from every .Net language. **All** these **types inherit** **from** the common **Object** class, including all Value types!

## Common Language Specification (CLS)

Defines the requirements and spec of all .Net languages. Includes the data types, classes etc.

## Roslyn

The C#/.VB compiler

# Language

## General

Built-In Aliases (String vs string):

Rather than using the type name or the fully-qualified type name, the built-in aliases for these types should always be used: bool, byte, char, decimal, double, short, int, long, object, sbyte, float, string, ushort, uint, ulong.

**Instead of ‘System.String’ or ‘String’, use ‘string’**

***Methods Signatures***

* In the context of method overloading, the signature of a method does not include the return value.
* But in the context of delegates, the signature does include the return value.
* From experimenting, I see that the signature does not include the static modifier (verified with delegate and overloading)

## Reference vs Value Types

|  |  |  |
| --- | --- | --- |
|  | Value Types | Reference Types |
| Description | Hold the value.  **Cannot be null** | A reference to the object that holds the actual value. |
| Types | Struct, enum, bool, numeric values, char. DateTime (struct) | string, array, all classes, interfaces, objects and delegates. |
| Memory |  |  |
| Copy | Copy value | Copy the reference only not the object (shallow). |
| Comparison | Compare value | Usually, compare only the reference, not the object (shallow). strings and delegates – deep comparison (of the object on the heap). |
| Check for empty | IsEmpty property | == null |

## Numerics

* Decimal – 128bit decimal values. Has more precision and a smaller range than floating point. For constant decimal values – use postfix of **m** or **M**

## Data Types

### Strings

#### Boxing

To avoid the performance hit (around 30% more in benchmarking test) of boxing/unboxing when building strings, always use .ToString() when including a non-string object inside a string.

For example:   
Console.WriteLine($”The result is: {result.ToString()}”);

#### Formatting:

string.Format(“string with parameters: {0}”,param1);

#### Concatenation

For one-liners, the best was to concatenate is:  
str = str1 + str2 + str3;

For builder a string over multiple calls:  
StringBuilder sb = new StringBuilder();  
sb.Append(str1);  
sb.Append(str2);  
sb.Append(str3);

#### Composite Formatting

The .NET composite formatting feature takes a list of objects and a composite format string as input. A composite format string consists of fixed text intermixed with indexed placeholders, called format items, that correspond to the objects in the list. The formatting operation yields a result string that consists of the original fixed text intermixed with the string representation of the objects in the list.

Example:

Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);

#### $ - string interpolation

This feature is available in C# 6 and later versions of the language. When available, it is more readable than the Composite Formatting:

Console.WriteLine(**$**"Hello, **{name}**! Today is {date.DayOfWeek}, it's **{date:HH:mm}** now.");

#### @ - string verbatim

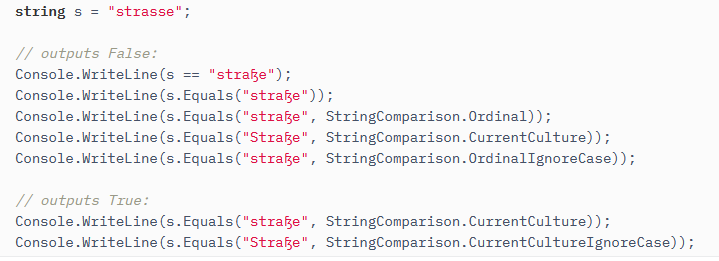
Used to define a verbatim string literal that doesn’t require escaping special characters (like raw string in Python). For example:

string filename1 = **@**"c:\documents\files\u0066.txt";

string filename2 = "c:\\documents\\files\\u0066.txt";

#### Comparing

* using == and .Equals(sting2) are the same.
* Use .Equals(string2, comparisonType) for example:



### Arrays

**int[] myArray** = {1, 2,3 4, 5};

myArray**.Length**

Pass array as argument:

MyFunction(new int[] {1, 2,3 4});

#### Array vs. ArrayList

* Array stores the values or elements of same data type but arraylist stores values of different datatypes.
* Arrays will use the fixed length but arraylist does not uses fixed length like array.

### List<T>

In general, it’s better to use lists in C# because lists are far more easily sorted, searched through, and manipulated in C# than arrays.

That’s because of all of the built-in list functionalities in the language.

### Dictionary<TKey, TValue>

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0>

A dictionary with unique keys that’s implemented as a Hash Table.

The performance depends on the quality of the hashing algorithm for the TKey’s type.

* Add/retrieve value: O(1)

You can use the default comparator, or supply your own (IEqualityComparer derived class).

### Lookup<TKey, TElement>

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.lookup-2?view=net-5.0>

Similar to Dictionary but with multiple values per key.

* Lookup<> objects are immutable - you can’t add/remove elements
* There is no public constructor for Lookup
* You can create a Lookup object by calling ToLookup() method on an IEnumerable object.

### Classes

#### Glossary

**Overriding** – virtual methods in derived class. Require the keyword **virtual** or **abstract** in the base class and **override** in the sub-class.

**Overloading** – multiple methods with the same name and different signatures in the same class. Does not require any special keywords, calculated automatically by the compiler.

**Finalize** – the destructor of the class. Will be called automatically by the garbage collector when the object is freed. Can’t be called explicitly in the code.

#### Access Level:

All the members and methods are defined as **private by default**.

Also supports:

* public
* protected
* internal – in the same Assembly (compilation unit) only
* protected internal

#### Polymorphism

In order to create polymorphism in C#:

* + - 1. Base class needs to define all its polymorphic methods as ‘virtual’:  
         public class BaseClass {  
          public **virtual** void VirtualFuncName(){…}  
         }  
         The ‘virtual’ keyword tells the compiler to add this method to the class’s virtual table. This means that whenever a derived class override this method, it will update the pointer for every object from the derived class.
      2. Derived class needs to define all its polymorphic methods as ‘override’ to mark them as overriding the base class’s virtual methods:  
         public class DerivedClass : BaseClass {  
          public **override** void VirtualFuncName(){…}  
         }  
         Note: the ‘override’ actually tells the compiler to update the method pointer in the virtual table to point to this method instead. So whenever we initialise a new DerivedClass, it will update this method pointer.  
         If we wouldn’t have included the ‘override’ keyword - the pointer would not have been updated. So when we call the method with a BaseClass object, it would have still pointed to the parent’s virtual method.

Notes:

* <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual>
* By default, methods are non-virtual. You cannot override a non-virtual method.
* You cannot use the virtual modifier with the static, abstract, private, or override modifiers.
* You can use the virtual modifier to define a virtual property:  
  …  
  public **virtual** int Number

{

get { return \_num; }

set { \_num = value; }

}

#### Abstract Classes

Classes that are the **basis for inheritance** but will never be used directly to create objects. Can **contain both concrete and abstract functions** and member variables.

If I have a default implementation that’s relevant to all the sub-classes, it makes sense to use abstract class. If I only want to define a ‘contract’ that all sub-classes need to support, I will use an interface.

Differences from interface:

* Can contain members and implementation
* Sub-classes can inherit from only **one abstract class**
* Can’t be initialised.

Note: Polymorphism works in abstract classes exactly like it would in interfaces and every **abstract function is also a virtual function!**

public **abstract** class Control  
{  
…  
// Abstract method -   
// just the signature without any implementation.  
// abstract functions must be overridden in the inheriting sub-classes.  
public **abstract** void SomeFunction(int x);   
}

#### Interfaces

Interface is a special type of class that defines a common interface.  
It differs from class definitions in the following ways:

* + Interfaces can contain **methods**, **properties**, **events**, and **indexers**. The interface itself does not provide implementations for the members that it declares. The interface merely specifies the members that shall be supplied by classes or structs that implement the interface.
  + An interface **cannot contain** *constants, fields, operators, instance constructors, finalizers, or types*, nor can an interface contain *static members* of any kind.
  + **All members are implicitly abstract**; Don’t include the abstract keyword
  + **All members are implicitly virtual**; Don’t include the override keyword.
  + **The class** itself as well as **all members are public** and can’t be changed; Don’t include the public keyword.
  + Interfaces can inherit from other interfaces in order to expand them.
  + Every class/struct can implement any number of interfaces but it can inherit from only one class.
  + A class that implement an interface, the implementation is sealed by default.  
    If you want to implement it virtually, you have to put the keywords ‘abstract’ or ‘virtual’ to allow to virtually inherit from this class

interface IEnumerator

{

// properties - define which methods have to be defined

// in any implementing class/struct

int SomeProperty { set; get; }

bool MoveNext();

object Current{get;}

}

#### Static Class

public static class StaticClass {  
 …  
}

A class that **all its member** methods and variables are **static**.  
Useful for **utilities classes**.

* + - They contain only static members
    - They can’t be instantiated
    - They are inherently sealed.

#### Sealed Class

sealed class SealedClass

{

}

A Sealed class cannot be inherited from.

Recommended for classes that

* + - Won’t be derived from
    - Consist of nothing but static methods and properties
    - Usually are not used by most programmers (more relevant for library writers)

#### Inheritance

* + - All classes in C# implicitly inherit from System.Object.
    - Derived classes can have only one direct base class. This include concrete or abstract classes.
    - Derived classes can implement any number of interfaces.
    - A derived class gain all the members of the base class, except for:
      * The base class’s constructors
      * The base class’s finalizers.
    - A class can prevent other classes from inheriting it or from inheriting any of its members by defining itself/its members sealed.
    - The base keyword is used to access members of the base class from within a derived class:
      * Call a method on the base class that has been overridden by another method:

…

public override void GetInfo()

{

// Calling the base class   
 // GetInfo method:

**base**.GetInfo();

…

}

* + - * Specify which base-class constructor should be called when creating instances of the derived class:

// This constructor will call

// BaseClass.BaseClass(int i)

public DerivedClass(int i) **: base(i)**

{

}

#### Extension Methods

Allows us to extend any type with additional methods, even types that we didn’t create ourselves!

static class StringExtensions  
{

// This function extends the string class so that we can call 'Shout' on any string.   
// Note: the static is mandatory and so is the 'this' before 'string' which is the class we're extending.

public static string Shout(this string s)  
{  
 return s.ToUpper() + "!!!";  
}

}

**CAUTION:** Microsoft recommend creating additional extension methods only when there’s no other choice since they can break in future .Net versions. For example: in the above example, if MS will release a new .Net version with Shout method implemented in string, this will break our own string.Shout extension.

In real life, we usually consume (e.g. linq methods) rather than create extension methods.

#### Member Variables == Class Fields

Fields are ordinary member variables or member instances of a class.

#### Properties == Accessors

Properties are an abstraction to get and set their values. Properties are also called accessors because they offer a way to change and retrieve a field if you expose a field in the class as private. Generally, you should declare your member variables private, then declare or define properties for them.

A value of a class (can be a concrete member or just a computable value) that can be set and/or get from the outside world.

* + he [get](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/get) and [set](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/set) portions of a property or indexer are called accessors. By default these accessors have the same visibility or access level of the property or indexer to which they belong.
  + You can use accessor modifiers only if:
    - the property or indexer has both set and get accessors.
    - The modifier is permitted on only one of the two accessors.
    - The accessibility level on the accessor must be more restrictive than the accessibility level on the property or indexer itself.
  + You cannot use accessor modifiers on an interface (public automatically) or an explicit [interface](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface) member implementation.  
    However, if the interface only defines one accessor, you can define the other one in the implementation with any access modifier.

Definition:

Class MyClass

{

public int PublicProperty

{

[protected/internal/private] get {return \_someValue;}

[protected/internal/private] set {\_someValue = value;}

}

// This is how you define auto-properties - the default getter  
 // and/or setter (just assignments with not processing:  
 // \*\* Use **prop <tab> <tab>** to auto create the code

Public int DefaultPublicProperty {[internal/private] get;set;}

}

**Auto-Property Initializers (from C# 6 and up):**

public int DefaultPublicProperty {[internal/private] get;set;} = initialValue;

for example:

public Collection<Something> Things {get; private set;} **=** new Collection<Something>();

#### Read-Only Auto-Property

If you don’t include a set in the property definition, it will be a read-only property which can be set only in the class’ constructor:

public int ReadOnlyProperty { get; }

readonly field – like const with the following differences:

* **Can be assigned in class’ constructor**
* **Can be assigned to non-const value in field declaration**:  
  public static readonly uint timeStamp =  
   (uint)DateTime.Now.Ticks;

**Const** value must be **initialized during its declaration to a constant value** (done in compilation time and not in runtime). In addition, every const variable will be declared implicitly as static.

**Static**

Members and methods like in C++ but can be accessed only from the class and not from objects! For example:

Var obj = new SomeClass();  
obj.SomeStaticFunction();

Will fail compilation. Instead you must access the static member/function through the class name directly:

SomeClass.SomeStaticFunction();

* Private static methods can be used to improve performance when critical (called with class vs objects)

#### Indexers

Indexers are similar to properties. They enable indexed properties that can be referenced using the array access syntax: somObject[index]

Example:

public class SomeClass {

// definition example:

public int **this[**string key[,optional additional params]**]**

{

get { return storage.Find(key); }

set { storage.SetAt(key, value); }

}

}

…

// usage example:

SomeClass someObject = new SomeClass();

var item = someObject[“key”];

someObject[“AnotherKey”] = item;

* indexes have the same rules as properties regarding access modifiers, inheritance etc.
* common usage:
  + accessing data that’s not available in memory as if it is while hiding the requires disk/network access from the user (e.g. historical DB records)
  + Modelling Dictionaries (1 or more dimensions)

#### Object Initializers

In order to initialize a new object using the default (public) constructor, you can supply initialization for all the public properties in an argument-like list:

Student student = new Student()  
{  
 FirstName = "Craig",  
 LastName = "Playstead",  
 ID = 116  
};

#### Reflection

Can be used to get information on types and Assemblies

Reflection has the following applications −

* + - It allows view attribute information at runtime.
    - It allows examining various types in an assembly and instantiate these types.
    - It allows late binding to methods and properties
    - It allows creating new types at runtime and then performs some tasks using those types.

Other uses for Reflection include constructing symbol tables, to determine which fields to persist and through serialization.

* + - There are System.Reflection is the namespace for the reflection. System.Reflection namespace defines the assembly module, MemberInfo, PropertyInfo, MethodInfo, ConstructorInfo, FieldInfo, EventInfo etc.
    - The System.Type is the base class for reflection. System.Type find the type name, namespace, module type etc. System.Type is also an abstract class.

// Using GetType to obtain type information:

Type type = i.GetType();

// Using Reflection to get information of an Assembly:

Assembly info = typeof(int).Assembly;  
Console.WriteLine(info);  
// The output is: System.Private.CoreLib, Version=4.0.0.0,

// Culture=neutral, PublicKeyToken=7cec85d7bea7798e.

// **dynamically load assembly** from file Test.dll

Assembly testAssembly = Assembly.LoadFile(@"c:\Test.dll");

// get type of class Calculator (in Test namespace)

// from just loaded assembly

Type calcType = testAssembly.GetType("Test.Calculator");

// **create instance of class Calculator**

object calcInstance = Activator.CreateInstance(calcType);

// get info about property: public double Number

PropertyInfo numberPropertyInfo = calcType.GetProperty("Number");

// **get value of property**: public double Number

double value = (double)numberPropertyInfo.GetValue(calcInstance, null);

// **set value of property**: public double Number

numberPropertyInfo.SetValue(calcInstance, 10.0, null);

#### Dependency Injection

<https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection>

1. Instead of depending on a concrete class, depend only on interfaces
2. Create a constructor that accept the required interface instead of initiating the objects in the class.

You can initialise the dependencies from your own context-code or use .Net’s dependency injection framework to initialize all classes and their expected initialization on program startup - see <https://www.tutorialsteacher.com/core/dependency-injection-in-aspnet-core>

Example:

public interface IMessageWriter

{

void Write(string message);

}

..

public class MessageWriter : IMessageWriter

{

public void Write(string message)

{

Console.WriteLine($"MessageWriter.Write(message: \"{message}\")");

}

}

…

public class LoggingMessageWriter : IMessageWriter

{

private readonly ILogger<LoggingMessageWriter> \_logger;

public LoggingMessageWriter(ILogger<LoggingMessageWriter> logger) =>

\_logger = logger;

public void Write(string message) =>

\_logger.LogInformation(message);

}

…

public class Worker : BackgroundService

{

private readonly IMessageWriter \_messageWriter;

public Worker(IMessageWriter messageWriter) =>

\_messageWriter = messageWriter;

…

}

…

class Program

{

static Task Main(string[] args) =>

CreateHostBuilder(args).Build().RunAsync();

static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureServices((\_, services) =>

services.AddHostedService<Worker>()

.AddScoped<IMessageWriter, MessageWriter>());

}

### Structs

For representing lightweight objects. Can save space compared to class objects for large objects collections.

The differences between classes and structs are:

* + Structs are value types and can’t be inherited from.

### Var

You can use the ‘var’ keyword instead of declaring the type of an object to have the C# compiler automatically assign the right type to the object. This can help make the code more readable. For example:

Instead of

Dictionary<string, Customer> dict =   
 new Dictionary<string, Customer>();

We can write:

Var dict = new Dictionary<string, Customer>();

From Microsoft Coding Conventions:

* + Use [implicit typing](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables) (var) for local variables **when the type** of the variable **is obvious** from the right side of the assignment, **or when** the precise type is **not important**.
  + Use var in for(…) loop: for(var i=0 ;…)
  + Don’t use it in forEach(..) loop: foreach(char ch in laugh)

### Anonymous Types

* + - Often preferable to tuples which don’t have named fields.
    - Anonymous types contain one or more **public read-only properties**.
    - No other kinds of class members, such as methods or events, are valid.
    - The expression that is used to initialize a property cannot be null, an anonymous function, or a pointer type.
    - Commonly used to organise data returned from queries:  
      var productQuery =

from prod in products

**select new** { prod.Color, prod.Price };

foreach (var v in productQuery)

{

Console.WriteLine("Color={0}, Price={1}",   
 v.Color, v.Price);

}

The C# compiler allows us to define anonymous types without explicitly declaring a class:

**var x = new {** Author = “Mark Seemann”, Title = “Dependency Injection in .Net”**};**

var y = new { Author = “Martin Fowler”, Title = “Patterns of Enterprise Architecture”};

var books = new[] {x, y};

The C# compiler, actually define the right class for x. When it sees ‘y’, it knows to use the same already defined class. And when it sees ‘books’, it knows to create the correct type of array of this anonymous class.

If we then define:

var z = { Author = “Robert Martin”, Title = “Clean Code”, Pages = 245};

The compiler will not use the previously created anonymous type but will create a new type. So, we can’t add ‘z’ to the books array we created before.

With anonymous types, we don’t have to supply the properties names. Instead, C# can infer them from the values. For example:

var author = “Adam Nathan”;

var title = “WPF 4”;

var book = new {Author = author, Title = title }; // regular anonymous type declaration

var book = new(author, title}; // inferred property names

**Note**: when using inferred property names, the C# compiler will name the properties exactly like the names of the variables we pass through, in this case, ‘author’ and ‘title’.

### Nullable Types

Nullable types are instances of the [System.Nullable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.nullable-1) struct. Nullable types can represent all the values of an underlying type T, and an additional [null](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/null) value. The underlying type T can be any non-nullable [value type](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types). T cannot be a reference type (which already has a null value).

You use a nullable type when you need to represent the undefined value of an underlying type. A Boolean variable can have only two values: true and false. There is no "undefined" value. In many programming applications, most notably database interactions, a variable value can be undefined or missing. For example, a field in a database may contain the values true or false, or it may contain no value at all. You use a Nullable<bool> type in that case.

* + **T?** is **shorthand for Nullable<T>** - which defines a variable of type nullable T. For example:  
    **int? x;** is equal to Nullable<int> x;
  + Use the [null-coalescing operator](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-coalescing-operator), **??**, to assign a value to an underlying type based on a value of the nullable type:

int? x = null;   
int y = x **??** -1;   
In the example, since x is null, the result value of y is -1.

So:

**<x> ?? <val>** is identical to :

**( <x> != null ) ? <x> : <val>**

* Nullable Type cannot be assigned into a value type. You will need to use ‘GetValueOrDefault()’ to assign it:  
  int? n = null;  
  int m = n.GetValueOrDefault();  
  GetValueOrDefault() - will return the value of the underlying type or, its default value if it’s null.

### Non Nullable Reference Types

Since C# 8.0, non-nullable reference types are supported to enable safer code.   
See: <https://dev.to/integerman/safer-code-with-c-8-non-null-reference-types-4f2c> for additional details and implementation.

### Common Standard Library Classes

#### IEnumerable<T>

Allow enumerating over a collection of objects.

Note: when we need to enumerate over a collection but we don’t need to use any other collection-specific functionality (e.g. List.Add), we should try to pass only the interface that we need. In this case, IEnumerable. This will allow us to change the type of collection we use without effecting the code that uses its IEnumerable interface!!

To **return an empty collection** - use this:  
**Enumerable.Empty**<T>();

#### Dictionary<TKey, TValue>

Holds key-value pairs.

O(1) to retrieve values (from key).

## Casting

* + var var1 = (SomeType)someObject;

Will throw an exception if the casting fails.

* + var var1 = someObject as SomeType;

Will set var1 to null if the casting fails.

### Casting vs As



In Bill Wagner’s [Effective C#](http://www.amazon.co.uk/Effective-Specific-Ways-Improve-Your/dp/0321245660/ref=sr_1_1?ie=UTF8&s=books&qid=1270477377&sr=8-1) book, he recommends that you should use the as keyword whenever possible, because:

* + it’s more efficient at runtime, the as keyword (like the [is](http://msdn.microsoft.com/en-us/library/scekt9xw(VS.71).aspx) keyword) does not perform any user-defined conversion. It will only succeed if the object is of the sought type (or derived from it) and never create a new object to satisfy a request.
  + requires less code because you don’t need a try-catch block in addition to a null check.

There are a few things you should keep in mind when using the as keyword:

* + it doesn’t work with value types because value types can never be null
  + don’t use the is keyword if you’re using as for type conversion, it’s redundant because these two statements are equivalent except the as version evaluates expression only once!

|  |  |
| --- | --- |
|  | expression as type  expression is type ? (type) expression : (type) null |

When you **should use casting**:

* + To work with custom conversions

## Operators

### ?. and ?[] – null-Conditional Operators

Used to test for null before performing a member access (?.) or index (?[]) operation. These operators help you write less code to handle null checks, especially for descending into data structures.

int? length = customers?.Length; // null if customers is null

Customer first = customers?[0]; // null if customers is null

int? count = customers?[0]?.Orders?.Count(); // null if customers, the first customer, or Orders is null

The **null-condition operators are** **short-circuiting**. **If one operation** in a chain of conditional member access and index operation **returns null**, then **the rest** of the chain’s execution **stops** **and** the whole expression **returns null**. In the following example, E doesn't execute if A, B, or C evaluates to null.

A?.B?.C?.Do(E);

A?.B?.C?[E];

Another use for the null-condition member access is invoking delegates in a thread-safe way with much less code. The old way requires code like the following:

var handler = this.PropertyChanged;

if (handler != null)

handler(…);

The new way is much simpler:

PropertyChanged**?.Invoke**(e)

The new way is thread-safe because the compiler generates code to evaluate PropertyChanged one time only, keeping the result in a temporary variable.

You need to explicitly call the Invoke method because there is no null-conditional delegate invocation syntax PropertyChanged?(e)

### => Lambda Operator

1. As a **lambda operator** in a lambda expression, it separates the input variable from the lambda body for example:

int shortestWordLength = words.Min((string w) => w.Length);

You can specify the type of the input variable explicitly or let the compiler infer it; in either case, the variable is strongly typed at compile time.

2. In an expression body definition, it separates a member name from the member implementation. For example:



### Printing

* + Console.log – print to console
  + Trace.log – print to output console for debugging on release version.

## Using

IDisposable interface - contain one method for disposing an unmanaged resource like files, streams, DB connections etc.

using (Font font1 = new Font("Arial", 10.0f))

{

byte charset = font1.GdiCharSet;

}

When the lifetime of an IDisposable object is limited to a single method, you should declare and instantiate it in the using statement. The using statement calls the [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) method on the object in the correct way, and (when you use it as shown earlier) it also causes the object itself to go out of scope as soon as [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) is called. Note: Dispose will also be called correctly if an exception occurs in the ‘using’ block. Within the using block, the object is read-only and cannot be modified or reassigned.

## Functions

* + There are no inline functions in C#. The compiler will always generate a function and a function call!
  + **ref parameters** - indicate that the parameter is initialized before calling the method and can be changed wi
  + thin the method. We can change a value parameter like this:  
    public void UpdateValue(ref int d){… d++}
  + **out parameters** - indicate that the parameters are output parameters. They are treated as reference types but don’t have to be initialised before calling the method. They do however have to be initialised within the method:  
    public void InitValue(out int d){… d = 1;…}

## Attributes

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/>

* + A method of associating metadata (information about types and members etc) and declarative information with code
  + Attributes can accept arguments
  + After an attribute is associated with a program entity, it can be queried at run time using reflection.
  + General Syntax:

**[**attributeName(optional params..)**]**  
<declaration of the entity to which the attribute applies>

Example:



* + All Attributes support two names: with an ‘Attribute’ suffix and without it.   
    For example: [Serializable] and [SerializableAttribute] are identical.

## Generics<T>

Defining a template of a class or method that can be declared and/or initialised with different types by the client code, without having to deal with boxing/unboxing or to do unsafe casts.

* + Usage: collection classes (e.g. List<T> etc)
  + The .NET Framework class library contains several new generic collection classes in the [System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic) namespace. These should be used whenever possible instead of classes such as [ArrayList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist) in the [System.Collections](https://docs.microsoft.com/en-us/dotnet/api/system.collections) namespace.
  + General format:

Public Class GenericList**<T>**  
{ …. }

var numbers = new GenericList**<int>**();

Coding standard: the type of the generics should be T[parameter meaning] to make it more readable like in the following example:

public class GenericDictionay<**TKey, TValue**>{…}

* + Dictionary<TKey,TValue> - dictionary of keys-value pairs. Implemented as a Hash Table.
    - (Try)Add, ContainsKey, TryGetValue, Remove etc.
    - The Keys must be unique.
    - O(1) – to retrieve a value from a key.
    - SortedDictionary<Tkey,TValue> - sorted on the keys
      * Binary Search Tree
      * O(log n) insertion, removal, retrieval
  + HashSet<T> - represent a set of values with no duplicates and no order.
    - Add, Remove, Contains, IntersectWith, IsProperSubsetOf, UnionWith etc.
    - Linq provide the following Set operations on any IEnumerable or Iqueryable objects: Union, Intersect, Except, Distinct
    - SortedSet<T> - a sorted set. Same performance as HashSet for inserting/removing items.
  + List<T> - single-linked strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.
    - Support [index]
    - **Better performance and type-safety than ArrayList**
    - SortedList<T>
      * O(log n) – retrieval
      * O(n) – insertion/removal
      * Use less memory than SortedDisctionary
    - SynchronizedCollection<T> - thread-safe list.
  + LinkedList<T> - a doubly-linked list
    - Insertion/removal, count - O(1)
    - Doesn’t support [index] access.
  + Queue<T> - FIFO collection
    - Enqueue, Dequeue, Peek etc.
  + Stack<T> - LIFO collection.
    - Push, Pop, Peek etc.

### Constraints

When we need to limit the type of parameters in the generic:

public T Max<T>(T a, T b) **where T : IComparable** {…}

The main types of constrains:

* T support specific interface: **where T : <interface name>**
* T is of class X or any of its derived classes: **where T : <base class name>**
* T is of value type (non reference): **where T : struct**
* T is of reference type: **where T : class**
* T has a default empty constructor: **where T : new()**

## Boxing

***Boxing*** is the process of converting a value type to an object in C#. This process can be explicit or implicit.

For example:

List<object> mixedList = new List<object>();

mixedList.add(“This is a string”);

mixedList.add(5); // implicit boxing occur

mixedList.add((object)3); // explicit boxing occur

**Boxing can be implicit or explicit; Unboxing is always explicit** and it’s the process of extracting the value from the object.

For example:

Object o = 5;

int i = (int)o; // explicit unboxing

**NOTE:** Every time you assign a value to an object-type, you’re implicitly boxing the value in an object. For example:

// 42 and true will be boxed in order to concet them to the string:  
Console.WriteLine(String.Concat("Answer", 42, true));

**Performance:**

Boxing and unboxing are expensive operations and therefore, we would want to use them only when we have to!

Since boxing created an object, it allocates the object on the heap, copy the value type into it and create a reference to it (value types exist on the stack) only).

## Serialization

Serializing objects into stream and recovering them allow us to save the object’s data in DB, File or memory or to transfer it to a remote application (e.g. in web), including passing it through different domains and firewalls.

* To mark that a class can be serializable, you have to add the SerializableAttribute to the class:

…

[**Serializable**()]

public class TestSimpleObject {

…

}

* + This class cannot be inherited from
  + All members of this class must be serializable (either classes with the SerializableAttribute or base types)
  + If you want to control how the class is being serialized, you may implement the ISerializable interface. In any case, you must also include the SerializableAttribute.

### JSON Serialization

[JSON Serialization](https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-text-json-how-to?pivots=dotnet-6-0)

* commonly used in web applications
* Serialize the public properties of the object into a string, byte array or stream
* The JSON output is minified by default (whitespace etc removed)
* To synchronously serialize an object (weatherForecast in this example):

string jsonString = **JsonSerializer.Serialize**(weatherForecast);

* Asynchrounouosly:

string fileName = "WeatherForecast.json";

using FileStream createStream = File.Create(fileName);

await JsonSerializer.SerializeAsync(createStream, weatherForecast);

await createStream.DisposeAsync();

* To pretty-print the JSON output, set [JsonSerializerOptions.WriteIndented](https://docs.microsoft.com/en-us/dotnet/api/system.text.json.jsonserializeroptions.writeindented#system-text-json-jsonserializeroptions-writeindented) to true:

var options = new JsonSerializerOptions { WriteIndented = true };

string jsonString =   
 JsonSerializer.Serialize(weatherForecast, options);

* You can use additional [JsonSerialiaztionOptions](https://docs.microsoft.com/en-us/dotnet/api/system.text.json.jsonserializeroptions?view=net-6.0) like IncludeFields, MaxDepth, IgnoreNullValues etc.
* Serialize to UTF-8 is 5-10% faster then using the string-based method but can’t be used to print out the result:

byte[] jsonUtf8Bytes =  
 JsonSerializer.SerializeToUtf8Bytes(weatherForecast);

* Deserialization:

WeatherForecast? weatherForecast =   
 JsonSerializer.Deserialize<WeatherForecast>(jsonString);

* Async deserialization:

using FileStream openStream = File.OpenRead(fileName);

WeatherForecast? weatherForecast =

await JsonSerializer.DeserializeAsync<WeatherForecast>(openStream);

* You can also deserialize a JSON string into a **JSON DOM** and extract only what you need from it.
* HttpClient and HttpContent used for http communication support an easy version of these operations using the web defaults for [JsonSerializerOptions](https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-text-json-configure-options#web-defaults-for-jsonserializeroptions). For example:

using HttpClient client = new()

{

BaseAddress = new Uri("https://jsonplaceholder.typicode.com")

};

// Get the user information.

User? user = await client.GetFromJsonAsync<User>("users/1");

…

// Post a new user.

HttpResponseMessage response =

await client.PostAsJsonAsync("users", user);

### Binary and XML Serialization

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/#binary-and-xml-serialization>

Warning: Binary serialization can be dangerous. Check security guidelines before proceeding!

## Delegates

Delegate represent **a list of** **reference to methods** with particular parameters list and return type (like C++ function pointers).

Delegates are used to pass methods as arguments to other methods. For example:

* + - passing a comparison method to a binary search method.
    - Event handlers

Example:

public delegate int Del(int i);

static void Main(string[] args)

{

Del myDelegate = x => x \* x;

int j = myDelegate(5); //j = 25

}

* + Delegates can contain **a list** of pointers to functions and when it’s called, it **will execute all the functions it’s pointing to one after the other**. For example:

myDelegate += function2;  
myDelegate += function 3;  
myDelegate(5) – will execute x\*x like in the previous example and then function 2 and then function 3.

**Note:** all delegates will be executed with the initial value (e.g. 5). The next function will not be executed with the result of the previous function!

#### Usage

Allows **decoupling the class that declare the delegate from the class that uses the delegate. Allows subscription-type functionality.**

Declare:

public delegate <return type> <DelegateName>(delegate parameters);

For example:

public delegate bool Compare(int num1, int num2);

Define the implementation:

Public class SomeClass  
{  
 Public bool MyCompare(int num1, int num2){…}  
}

Assign the delegate in the context code:

…  
SomeClass myObject = new SomeClass();  
var myDelegate = new Compare(myObject.MyCompare);

Use delegate:

myDelegate(5,7);

#### Built-In Delegates

* **Action/Action<>**

Build in C# delegate that **perform an action and returns void**. To use it we can write:  
Action<int, double> myActionDelegate =   
 <any function that takes an int and a double and returns void)  
myActionDelegate += <add more functions to the delegate>

* **Func/Func<>**

Build in C# delegate that calls **a function that returns a value**. To use it we can write:  
Func<int, double> myActionDelegate = <any function that takes an int and returns a double)

* You can use them (and other delegates) to pass a methods into functions. For example:

public void RegisterCalculation(Action<int, int> func)

{

binaryCalculator += func;

}

public void GetCalculatorForImmediateExecution(int x, int y, Action<int, int> func)

{

func?.Invoke(x, y);

}

* You can initialise/add to delegates with an existing method:

public int MyRandomGenerator1(){…}  
Func<int> getRandomNumber = MyRandomGenerator1;

* with a delegate anonymous function:

getRandomNumber += delegate(int x){…};

* with lmbda:

getRandomNumber += (int x) => {…};

#### Delegates vs. Interfaces

When to use delegates and when to use interfaces?

* + Use delegates when:
    - You use an event-driven design pattern
    - The caller doesn’t need to access other parts of the object implementing the function (e.g. data/methods)

## Events

* Fired (or raised) when “something happens” (e.g. button click)
* Implemented using delegates
* By convention (in .Net) always:
  + Return void
  + Have two parameters:
    - An object (the sender)
    - EventArgs or derived class (helper data)

#### Declare – in the event publisher:

// 1 - Define the delegate for the event handlers  
// convention: the delegate name should be <Event>EventHandler  
// for example:  
// public delegate void TimeChanged**EventHandler**(object obj, EventArgs e);  
// note; since .Net version 2, this is no longer needed and instead we can use .Net EventHandler or EventHandler<> types instead when  
// defining the event handlers list:

// 2- Define the event that will trigger the event handlers:  
// convention: the event name should be describe the event  
// public event TimeChanged**EventHandler** TimeChanged;

// EventArgs: either a built-in type or your own custom type to hold

// the event arguments.  
public event EventHandler<EventArgs> TimeChanged;   
  
// 3 – raise the event  
…  
 Create e… or send EventArgs.Empty instead of e  
 TimeChanged?.Invoke(this, e);

…

#### Implement the Event Handlers (in different subscriber classes):

void **On**TimeChanged(object obj, EventArgs e){…}

#### Subscribe to Event (in subscribers or Context Code):

theObject.TimeChanged += subscriber.OnTimeChanged;

## Lambda

A lambda expression is an [anonymous function](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/anonymous-methods) that you can use to create [delegates](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates) or [expression tree](http://msdn.microsoft.com/library/fb1d3ed8-d5b0-4211-a71f-dd271529294b) types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.

**We usually want to use Lambdas for short, one-line methods. If our Lambda expression is a few lines long, we probably want to refactor the code into a proper method.**

### Expression Lambdas

Creating a lambda expression:

**(input-parameters) => expression**

An expression lambda returns the result of the expression. You don’t need to indicate the parameters value or the return type.

delegate int del(int i);

static void Main(string[] args)

{

del myDelegate = x => x \* x;

int j = myDelegate(5); //j = 25

}

Creating an expression tree:

using System.Linq.Expressions;

namespace ConsoleApplication1   
{

class Program

{

static void Main(string[] args)

{

Expression<del> myET = x => x \* x;

}

}

}

### Statement Lambdas

Note: Statement lambdas, like anonymous methods, cannot be used to create expression trees.

=> : goes to operator.

**(input-parameters) => { statements that evaluate to a value; }**

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

* You don’t indicate the parameters types (although it is type-safe)
* You don’t need a return keywork since the lambda evaluate to a value.

### Lambda Scope

A lambda expression defined inside a function will have **access to all the other variables defined in the same scope (function/class).** In addition to the paramters it receives. For Example:  
…  
const int factor = 5;  
Func<int, int> multiplier = (n) => (n\*factor);

## Enumerables and Iterators

* Enumerable - every class that implements the IEnumerable and IEnumerator interfaces, allowing iteration over its collection and using Linq queries on it (e.g. for item in collection….select)
* To turn a class into an enumerable class, you must implement:
  + a class that implements IEnumerator<T> - implements all enumerator actiions: get Current, Reset enumerator, MoveNext etc.
  + your class needs to implement IEnumerable<T> interface - implement GetEnumerator (that returns an object from the previous IEnumerator class)
* Additional information on how yield works: <http://shadowcoding.blogspot.com/2009/01/yield-and-c-state-machine.html>

Using enumerable collections/classes with foreach:

[await] foreach (var item in collection)

{

Console.WriteLine(item?.ToString());

}

this is translated by the C# compiler to:

IEnumerator<int> enumerator = collection.GetEnumerator();

while ([await] enumerator.MoveNext())

{

var item = enumerator.Current;

Console.WriteLine(item.ToString());

}

* **foreach** enumerates the collection to supply each member in turn.
* You will use **await foreach** when waiting on an asynchronous enumerator.
* In order to support enumeration on your collection/class , you need to implement either the
  + IEnumerable<T> for synchronous processing

Example:

public IEnumerable<int> GetSingleDigitNumbersLoop()

{

int index = 0;

while (index < 10)

**yield** **return** index++;

}

or,

* + IAsyncEnumerable<T> for asynchronous processing  
    Example:  
    public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()

{

int index = 0;

while (index < 10)

**yield return** index++;

await Task.Delay(500);

yield return 50;

await Task.Delay(500);

index = 100;

while (index < 110)

yield return index++;

}

* + Note: instead of enumerating directly using the ‘yield return’ statement, you can also return a type that is enumerable by itself (like any collection type).
  + Note: you can’t have a ‘return’ statement in addition to the ‘yield return’ statements in Enumerable methods. If you need both, you might need to split them into two methods:

public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)

{

if (getCollection == false)

return new int[0];

else

return IteratorMethod();

}

private IEnumerable<int> IteratorMethod()

{

int index = 0;

while (index < 10)

{

if (index % 2 == 1)

yield return index;

index++;

}

}

## Asynchronous Programming

In windows 8 and above, the standard states that:  
 **everything that can take over 50ms must be done asynchronously!!**

Callback:

In old systems, callbacks were used to connect asynchronous calls from the main code.

These made for a buggy, hard to maintain code because they turn the code inside-out.

Callback State Machine:

* + - Since C# 5, a new **task based** model was introduced in which the framework transforms the code into a callback state machine by itself. In order to do this, you should:
      * Mark the asynchronous functions with the new ‘async’ modifier
      * Make sure that the asynch method returns Task<T>
      * User ‘await’ operator to yield control.
      * Your code resumes after the ‘await’ completes normally

### Define an Async Method

public **async** **Task**<returned type> DoSomethingLongAsync(<params>) {…}

to define an async method, we:

* + Add the async decorator to the definition
  + The function must return either Task or Task<> type. The Task class represents a single asynchronous operation.
  + Convention: the function should be called <Function name>Async
  + Note: **most C# library methods** that takes a long time (e.g. StreamWriter::Write or WebClien::Download) **has an Async version as well**.

### Use Async Methods

When calling an async method:

var html = **await** webClient.DownloadStringTaskAsync(url);

* + An **await** expression **does not block** **the thread** on which it is executing, **just the current async method**. Instead, it causes the compiler to sign up the rest of the async method as a continuation on the awaited task, similar to a callback that will be called automatically once the async operation completes. **Control then returns to the caller of the async method**. When the task completes, it invokes its continuation, and execution of the async method resumes where it left off.
  + An **await** expression **can occur only in** **async method**, lambda expression, or anonymous method. The term await serves as a keyword only in that context. Elsewhere, it is interpreted as an identifier.
  + In non-a
  + **To wait at the end of the method:**

var htmlTask = webClient.DownloadStringTaskAsync(url);  
….   
Code to run immediately after initializing the async call  
…  
// Now wait for the async call to finish:  
Var html = **await** htmlTask;

* + **Problem: How to call an async method from a sync method?**
    - **Solution:**

myObject.FunctionAsync()**.Wait();**

or

**Task**<int> asyncTask **=** myObject.FunctionAsync();

…

// note: this code will **execute in parallel** to the

// execution of the async function FunctionAsync();

// Only when the code encounters the .Wait() call –

// it will stop and wait for the function to complete.

asyncTask**.Wait();**

This will wait until the FunctionAsync is finished and thus can be called from a synchrous code.

* + - **Solution:**var task = Task.Run<>(async () =>   
       await FunctionAsync());  
      return task.Result;

This will run the asynchronous call on a different Thread, returning only when the 2nd thread run has finished.

* + **await vs Task.Wait():**
    - await - asynchronous - will put the rest of the method in the task to be executed once the call is finished. This will not block the execution on the thread.
    - Task.Wait() - synchronous call. The thread will block until the task has finished.
    - Use Task.WhenAll(…) to wait on multiple tasks:  
      **await Task.WhenAll**(  
       eggsTask, baconTask, toastTask);
    - Task.WhenAny() will return when any of the tasks return

### Async Design

* + If you have a combination of async and sync commands, you might want to wrap them together in an async method that will free your main thread while they process.

For example, instead of this code:

Toast toast = await toastTask;

ApplyButter(toast);

ApplyJam(toast);

Console.WriteLine("Toast is ready");

which pause the thread while waiting for the toast before spreading butter and Jam, you can group them together in an async method and only wait on it where it makes sense:

static async Task<Toast> MakeToastWithButterAndJamAsync(int number)

{

var toast = await ToastBreadAsync(number);

ApplyButter(toast);

ApplyJam(toast);

return toast;

}

### Async Exceptions

## Threads Management

Managed Thread Pool:

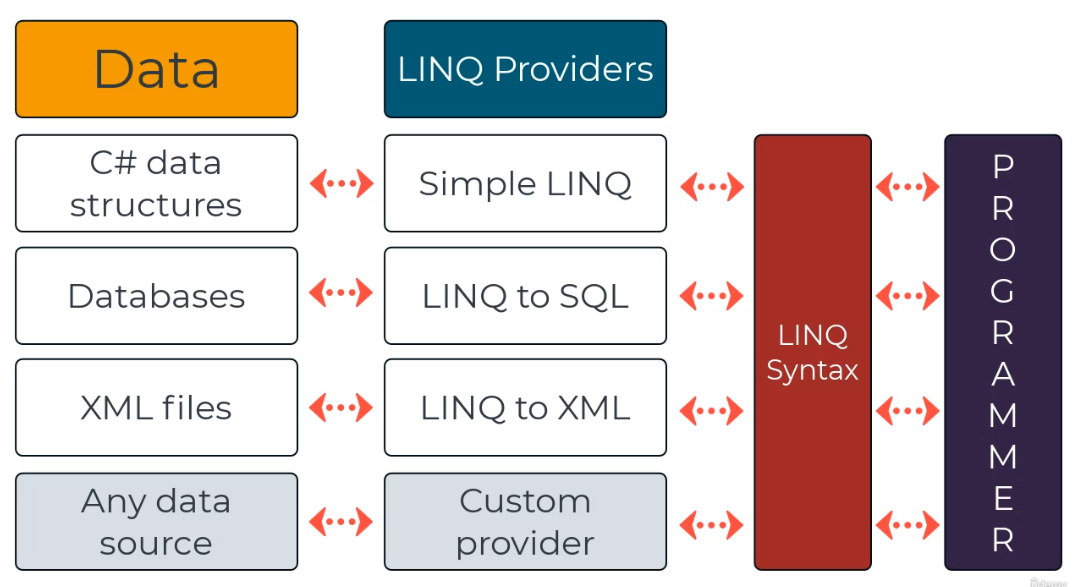
<https://docs.microsoft.com/en-us/dotnet/standard/threading/the-managed-thread-pool>

Thread pool threads are [background](https://docs.microsoft.com/en-us/dotnet/standard/threading/foreground-and-background-threads) threads. Each thread uses the default stack size, runs at the default priority, and is in the multithreaded apartment.

Once a thread in the thread pool completes its task, it's returned to a queue of waiting threads. From this moment it can be reused. This reuse enables applications to avoid the cost of creating a new thread for each task.

There is only one thread pool per process.

# LINQ – Language Integrated Query



.Net support different linq providers:

* LINQ to objects in memory (e.g. collections)
* LINQ to SQL (LINQ to Entities)
* LINQ to XML
* LINQ to ADO.NET Data Sets
* You can also write your own Linq provider (although it’s pretty rare).

Allow you to query any collection, including collections in memory.

**Get back:**

* A list of results or
* A collection of anonymous classes

There are two ways to create a LINQ syntax:

* **Query Syntax looks like SQL**:  
  var results = **from** x **in** booklist **where** x.Author = “Jesse Liberty” **select new**{x.Title,x.Author}
* **Method Syntax uses lambda expressions**:  
  var results = booklist**.Where**(x=>x.Author == “Jesse Liberty”);

**Three Stages:**

The three stages of a Linq query:

* + - 1. get the data source:  
         int[] number = new int[] {0, 1,2 3, 4};
      2. create the query -   
         Note: this only creates the query object, it doesn’t run it yet!!  
         The query execution is deferred until its data is actually needed!  
         var numQuery =  
          from num in numbers  
          where (num % 2) == 0  
          select num;
      3. Execute the query:  
         foreach (int num in numQuery) {  
          ….  
         }

Note: every query that returns an Enumerable has deferred execution. This means that you can define the query once in your code but then use it multiple times to get updated results (i.e. every time your execute the foreach loop).  
*\* similar to defining the Enumerator or the Regex once but then calling the matching methods multiple times.*

* My best understanding is that:
  + If the query returns an actual value - it executes immediately (like Count() or Max())
  + If the query returns an IEnumerable type, it defers the execution until the enumerable is used (e.g. in a foreach loop) - like zip() or take().  
    That’s why we can define the query one time and then use it multiple times to get the updated data or to refine the query.  
    For example:  
    var all300millionAmericans = allPeople.Where(person => person.Nationality == “USA”);  
    ….  
    var americansForOnePage = all300millionAmericans.Take(100);   
    PrintToPage(americansForOnePage); // this is where the query actually executes,   
     // on only 100 records instead of the full millions of them
  + Deffered Execution Under the hood: <https://itnext.io/linq-deferred-execution-explained-2844838f83ef>
  + Note: Linq **never modifies its input data**, it only returns a new IEnumerable or end value. This also true for linq methods like Append (since linq just returns IEnumerable and not an actual data structure)

**Forcing Immediate Execution:**

* Queries that perform aggregation functions over the resulting collection (e.g. Count, Average, Any, First etc) - any method that return one value from the collection - will run immediately (since they need to run the loop in order to get the value):

var maxEven =

(from num in numbers

where (num % 2) == 0

select num.Max();

* To run the query immediately and save the results, you can also call the ToList or ToArray at the end of the query:

List<int> numQuery2 =

(from num in numbers

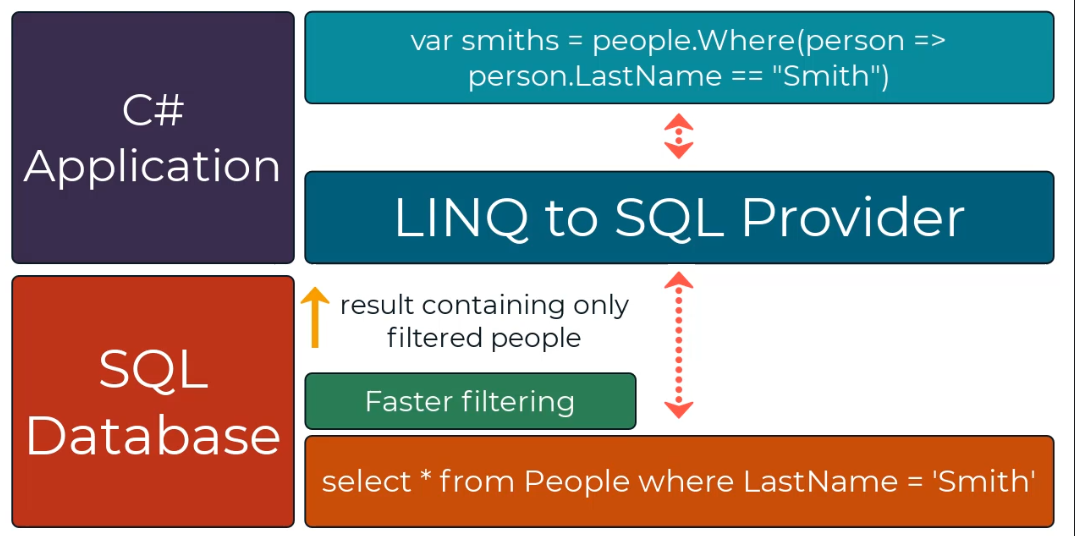
where (num % 2) == 0

select num).ToList();

**Internal details:**

* LINQ provides extension methods on IEnumerable<T>. This is what allows us to “pipeline” LINQ calls together.
* We can use anonymous types to pass state through LINQ pipelines.

## Performance

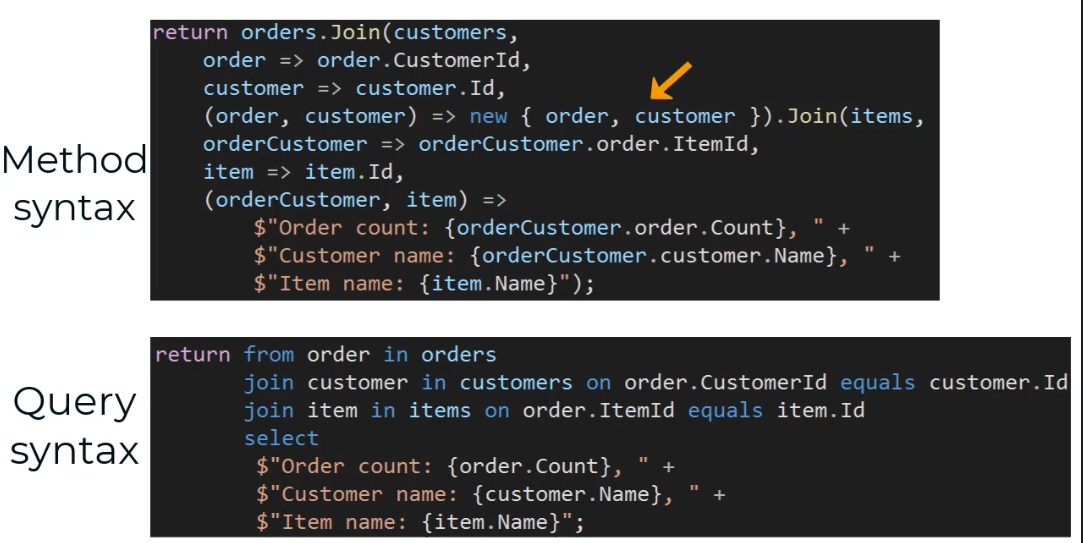
* Linq on SQL runs the query on the DB itself (much faster than in memory) and only return the results which makes it save memory and also run faster:  
  
* On data in memory - Linq is slower than manual iteration
* Linq joins are relatively faster than other methods (e.g. select). However, still (slightly) slower than using manual iterations (as long as they are optimised)
* <https://timdeschryver.dev/blog/make-your-csharp-applications-faster-with-linq-joins>

## Method Syntax Using LINQ Extension Methods

* The Methods Syntax is usually more powerful than the LINQ Query Operators. Some complex queries don’t have a keyword in the query operators but they do have an extension method we can use.

var cheapBooks = books  
 .Where(b => b.Price < 10)  
 .OrderBy(b => b.Title)  
 .Select(b => b.Title);

* The Query Syntax is sometimes more readable, for example when doing complex joins. For example:



* Sometimes, we’ll need to run a combination of both, for example, when we want to use the simplicity of the query syntax but we need to power of the Method Syntax. For example:

var cheapBooksWithNoDuplications =  
 (**from** b in books  
 where b.Price < 10  
 orderby b.Title  
 **select** b.Title).Distinct();

## LINQ Query Operators

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/query-keywords>

### Query Operators - Deferred Execution

* When uses only supported keywords, this syntax is usually more clear and readable.

var cheapBooks =  
 **from** b in books  
 where b.Price < 10  
 orderby b.Title  
 **select** b.Title;

* **Note:** LINQ query operators **always**:
  + **Start** with **from**
  + **End** with **select**

#### Aggregate

Execute a method on all elements in the collection while aggregating the intermediate result (like .map in JS).

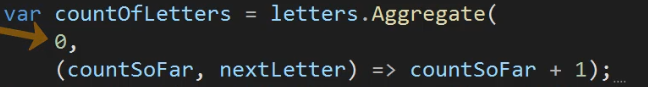
Aggregate takes a lambda of the form:   
(<intermediate result>, <current item>)

Example:

var sumOfNumbers = numbers

.Aggregate( (sum, nextNumber) => sum + nextNumber);

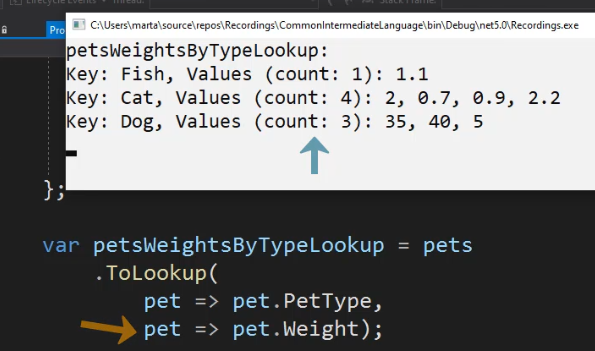
* We can set the initial value for the aggregated value by using the overloaded method, like this:



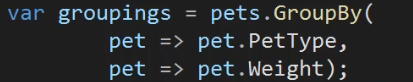
#### GroupBy

Creates a Lookup according to the supplied key(s).

Using Lookup, we will write:



Using GroupBy, we will write:



Both return the exact same Lookup.

Optional Parameters:

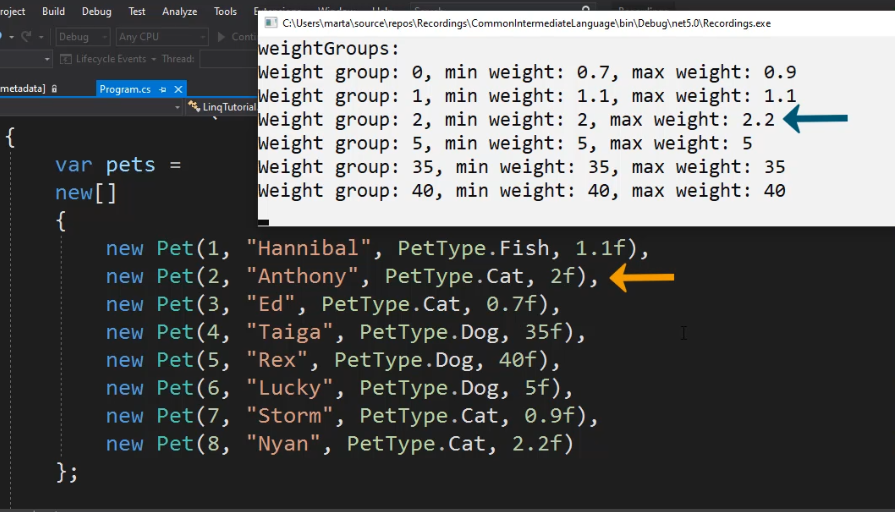
* **elementSelector**: [Func](https://docs.microsoft.com/en-us/dotnet/api/system.func-2?view=net-6.0)<TSource,TElement>   
  Allows us to select the elements in the Lookup list, or as the input items for the **resultSelector**. If missing, will use the whole object (e.g. pet)
* **comparer**: [IEqualityComparer](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.iequalitycomparer-1?view=net-6.0)<TKey>   
  Allow us to supply a comparer object for the key comparisons.  
  If missing, will use the default comparer for the key types.
* **resultSelector**: [Func](https://docs.microsoft.com/en-us/dotnet/api/system.func-3?view=net-6.0)<TKey,[IEnumerable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1?view=net-6.0)<TSource>,TResult>   
  Allow us to construct new anonymous objects as the items in the Lookup lists.  
  If missing, will use the elementSelector.  
  **Note**: when using resultSelector, the result of the query will be an enumerable of a **TResult** collection, not of a Lookup with keys and lists of items…

Additional Notes:

* When the GroupBy returns a Lookup with keys and value-lists (in all cases except for when we use the resultSelector), we can access each group’s key using the **Key** keyword



which results in:



#### Intersect , Except , SequenceEqual

* Intersect - return the objects present in both collections
* Except - return the objects in the first collection that don’t exist in the second collection.
* SequenceEqual - return true/false if the collections are equal, **including order**.
* Allows supplying customer Comparer for the equality check.

#### Join - Inner Join

* Performance - somewhat close to O(2(N+M)) - Uses Initializes and use Lookup to traverse the collections.
* Return only values that **exist in both tables**
* [Join](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.join?view=netstandard-2.1) preserves the order of the elements of outer, and for each of these elements, the order of the matching elements of inner
* We can use chaining to join more than two collections (join the first two, then join with the 3rd etc)
* When joining more than two collections, using the query syntax is more readable.

#### GroupJoin - Left Join

Will return all values in the **left table** with whatever exist on the right one.

It basically does a GroupBy the common key (in both tables) and return a Lookup similar to the GroupBy method.



#### Select –

* transform the collection to a different of collection by applying an operation to each member of the original collection. This also allow us to change the type of elements in the new collection.  
  Example:  
  var studentNames = students.**Select**(student =>   
   $”{student.Name}, {student.LastName}”);

#### SelectMany

the same as Select method with one different - SelectMany is used when we want to get a collection of items in each operation (rather then just one as in Select). SelectMany get all the collections and flatten them to get just a single-level collection of items.

Under the hood, this is just a nested forEach loop.

Example:  
var studentGrades = students  
 .**SelectMany**(student => student.Grades);

Under the hood, this translates into:

Enumerable <float> studentGrades = new Enumerable<float>();

forEach( var student in **students**)

{

forEach(var grade in **student.Grades**)

{

studentGrades.Add(grade);

}

}

* If we need to flatten even more levels, we simply use the SelectMany multipleTimes.
* To keep track of the outer collection while processing the inner collection, we can use another overloaded method that takes as a 2nd parameter a resultSelector that process both (outer, inner) values and produces the TResult value (like in Select):  
  var ownerPetPairsInfo = people  
   .**SelectMany**( person => person.Pets,  
   (person, pet) => $”{person.Name} is the owner of {pet.Name}”);
* We can also use SelectMany to traverse over multiple collections (that are not necessarily internal to each other). For example:

int[] numbers = {1,2 3};

int[] letters = {‘a’, ‘b’, ‘c’};

var combinedProduct = numbers

.**SelectMany**( number => letters,

(number, letter) => $”{number} - {letter}”);

#### Where

* – return all the elements in the collection that satisfy the condition
* **OrderBy/OrderByDecending** - for sorting the collection
* **group** – returns a sequence of [IGrouping<TKey,TElement>](https://docs.microsoft.com/en-us/dotnet/api/system.linq.igrouping-2) objects that contain zero or more items that match the key value for the group; returns a list of lists according to the key:

var studentQuery =

from student in students

group student by student.Last;

// will return a list of lists of students.   
// All the students in each sub-list will have the same last name (as key)  
// TKey is string (student.Last) and the TElement is Student (student)

#### Zip

combine two collections that doesn’t (necessarily) have a relationship between them into pairs.

Example:

var numbers = new int[] {1,2 ,3, 4,5} ;

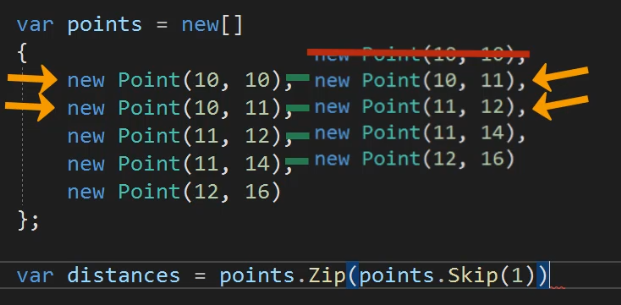
var names = new string[]{“Jhon”, “Steve”, “James”};

var combines = numbers.**Zip**(names);

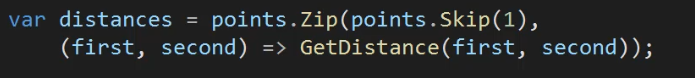
// output will be the following touples:   
// {(1,”Jhon”), (2,”Steve”), (3, “James”)}

// Note: the leftover values will be ignored

Sometimes we want to Zip a collection with itself. For example, zipping each value with the next value in the list:



Zip can also receive a 2nd resultSelector parameter:



### Aggregators on the Enumerable

<https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable?view=net-6.0#methods>

* **Append/Prepend** - add a new element to the end/start of the collection.  
  Note: the query will not change the original collection, just return the additional elements as part of the enumerator!
* **Any** – does the collection contains any items that match the criteria?
  + If there is no criteria - .Any() - will return if there are any elements in the collection
* **All** - all elements meet the criteria
* **Concat** & **Union** - concat two collections. union will remove duplicates in the process and concat will not.
* **contains** – does the collection contains an item that matches val:

bool doesContainValue = someCollection.Contains(val);

* **Distinct** - remove all duplicated elements. Note: for reference types, the reference is compared and not the values of the objects.
  + Distinct()
  + Distinct(IEqualityComparer<T>) - requires creating a compararer class ()
* **ElementAt** - access element at a specific index. Required for example when we treat a collection as an IEnumerable<t> since it doesn’t support the [] operator.
  + Note: ElementAt doesn’t check the index boundaries and will not issue a compilation error if using an out of bound index. Instead, it will throw a runtime exception
* **OfType** - collect all objects from the collection from the same type. This is used when working on collections that share the same interface.
* **Take/TakeLast** – return sub-set of the results
  + TakeWhile(predicate) - will take elements from the collection as long as the predicate is true. Once it hit false, returns (skipping the rest of the collection).  
    Usually best used with ordered collections (allow short breaking once we reached the condition)
* **zip** – works on two collections in parallel – for combining or manipulation.
* **Single** – return the only element in the collection that satisfy the condition. If there isn’t exactly one element like this, throws an exception
* **SingleOrDefault** – Like single with the following difference: if there isn’t any element that matches the condition, will return a default value. If there are more than one element, will still throw an exception.
* **First/FirstOrDefault ; Last/LastOrDefault**
* **Skip, SkipLast** and **SkipWhile** - commonly used with take:  
   Skip(num).Take(num) – used for paging.
* **Where** - filter the collection based on a predicate. If no item in the collection matches the predicate, it will return an empty collection and won’t throw an exception.
  + Has also an overload that except both the item and its index in the collection:   
    myCollection.Where( (item, index) => {…});

### Collection Types & Change

* OfType<type> - filter a collection (can be of objects) according to their type. Example:  
  var strings = objects.OfType<string>();
* .ToArray()
* .ToList()
* .ToHashSet() - for unique values
* .ToDictionary() - for unique keys
* .ToLookup - for immutable Lookup with multiple values per key. Note: this is very inefficient when working with databases. In this case, it would be better to use the GroupBy() method instead that translates into SQL’s group\_by query.
* .AsEnumerable() - will change the type from the derived type to IEnumerable itself. This is useful if we need to use the default implementation to some Linq query that was overwritten in an enumerable-derived type.
* Cast<toType> - to cast each element in the collection to a different type.

### Generating New Collections

**Enumerable.**

* .**Empty**<T>() - create an empty collection.  
  var emptyInts = Enumerable.Empty<int>();
* .**Repeat**(<value to repeat>, <the number of times to repeat>)
* Enumerable.**Range**(<starting value>, <the number of elements to add>) -  
  will return a collection of sequential values starting from the start value.
* .**DefaultIfEmpty**() - will return the original collection if it’s not empty. But, if it’s empty - will return a single element with the default value for the collection’s type. For example:

var emptyNumbers = new int[0];  
var defaultIfEmpty = emptyNumbers.DefaultIfEmpty();  
// will return {0} which is the default integer.

We can also supply the default value to use if the collection is empty.

For example:

var defaultIfEmpty = emptyNumbers.DefaultIfEmpty(97);

// will return {97}

### Performance

<https://linqsamples.com/tutorials/looking-into-linq-performance>

from: <https://itecnote.com/tecnote/c-what-guarantees-are-there-on-the-run-time-complexity-big-o-of-linq-methods/>

There are very, very few guarantees, but there are a few optimizations:

* Linq usually tries to use the most efficient method of the underlying types used in the collection so if we use efficient types, the implementation will be quicker than if we use wasteful types.
* Extension methods that use **indexed access**, such as ElementAt, Skip, Last or LastOrDefault, will check to see whether or not the underlying type implements **IList<T>**, so that you get **O(1)** access instead of O(N).
* The **Count** method checks for an **ICollection** implementation, so that this operation is O(1) instead of O(N).
* Distinct, GroupBy Join, and I believe also the set-aggregation methods (Union, Intersect and Except) **use hashing**, so they should be close to **O(N)** instead of O(N²).
* **Contains** checks for an **ICollection** implementation, so it may be O(1) if the underlying collection is also **O(1),** such as a HashSet<T>, but this is depends on the actual data structure and is not guaranteed. Hash sets override the Contains method, that's why they are O(1).
* **OrderBy** methods use a stable quicksort, so they're **O(N log N)** average case.

I think that covers most if not all of the built-in extension methods. There really are very few performance guarantees; **Linq itself will try to take advantage of efficient data structures** but it isn't a free pass to write potentially inefficient code.

# Dynamic

Allows defining and using dynamic variables like you would in python or javascript. For example:

**dynamic** name = “MyName”;  
name = 10;

This code would have failed as normal, static C# code but it will compile and run successfully as dynamic.

Dynamic is meant to be used when you need to use and communicate with dynamic languages or COM objects e.g. Word COM objects or javascript.

**NOTE**: The dynamic keyword is **not meant to replace any existing .NET-only code.**

When using dynamic, C# uses the DLR (Dynamic Language Runtime) (on top of the CLR) to run the code.

# Exception Handling

try  
{  
 // the code that can throw exception  
}  
catch (Exception ex)  
{  
 // handle exceptions, log them and/or throw them up for further handling  
 throw;  
}  
finally  
{  
 // Will run whether there was an exception or not -   
 // used to perform code cleanup.  
}

## Best Practices:

* Use try-catch blocks for code that might throw an exception
* Exception is an **expensive mechanism**! Therefore, handle common conditions without throwing an exception. The general Microsoft guidelines are:
  + **Use exception handling** **for exceptional events** - if the event doesn't occur very often, that is, if the event is truly exceptional and indicates an error (such as an unexpected end-of-file). When you use exception handling, less code is executed in normal conditions.
  + **Check for error conditions** in code if the event happens routinely and could be considered **part of normal execution**. When you check for common error conditions, less code is executed because you avoid exceptions.
* Design classes so that exceptions can be avoided. For example, classes that return some special value (e.g. EOF) instead of throwing an exception (e.g. EndOfFileReachedException).

## Catch

You can add as many catch blocks as you need, from the most specific to the most generic. You can also have a final generic catch-all catch block:

**catch**  
{…}

### Exception Filters

C# 6.0 and above support exception filters. They allow you have even more control over your catch blocks and further tailor how you handle specific exceptions. This can help you fine-tune exactly how you handle exceptions and which ones you want to catch. Example:

catch (WebException ex) **when** (ex.Status == WebExceptionStatus.ProtocolError)  
{..}

catch (WebException ex) when ((ex.Response as HttpWebResponse)?.StatusCode == HttpStatusCode.NotFound)

{

//code specifically for a WebException NotFound

}

## Finally

* If there was an exception – will run after the catch block.
* If there was no exception – will run after the try block.

Used to release unmanaged resources (i.e. resources that the CLR doesn’t manage and don’t have garbage collection) for example: file handlers, network handlers, graphic handlers etc.

#### Dispose

Defined in System.IDisposable and used for releasing un-managed resources. Note: Dispose does not replace an object’s destructor (finalizer ~ClassName) which is called automatically by the CLR when the garbage collector deletes the object.

**Note:** every class that manages unmanaged resource should implement the IDisposable interface.

To release the unmanaged resource:

unmanagedResource**.Dispose()**;

#### Using

Another, more elegant way to handle unmanaged resources release is with the using block:

**using**( StreamReader reader = File.OpenText(“file.txt”))  
{  
 .. the same code as in the try block  
}

reader.Dispose will be called automatically by the CLR whenever the using block exits whether it threw an exception or not.

The above code is completely identical to:

try  
{  
 StreamReader reader = File.OpenText(“file.txt”);  
 …  
}  
finally  
{  
 if(reader != null)  
 {  
 reader.Dispose();  
 }  
}

## Throwing Exceptions

Useful when you want to:

* **Add information** to a received error/exception or to change its resolution. For example, if you get an error/exception while trying to stream a video on your website, you might want to create a use a new YouTubeException and return it, instead of returning a more low level but less expressive NetworkException.
* **reduce information**: for example, if the exception contains secret information that we don’t want potential hackers to see, we can throw a higher/new level exception instead.
* When an **error is unrecoverable** and the program shouldn’t continue running if it occurs.

### New Exception Types

You can define new exception types (classes) by inheriting from Exception or one of its derivative classes:

public class YouTubeException : Exception   
{…}

### Throw Exception

…  
**throw new** YouTubeException(“error message to display”, **ex**);  
…

Note: when throwing an exception from within the catch block, it is usually a good idea to throw the exception we caught (e.g. ‘ex’ above) since it will allow us better debugging capability.

### Re-Throw Exception - Throw Clause

try {…}  
catch( Exception ex)  
{  
 …  
 **throw;**  
}

Note:

* Don’t use throw ex; otherwise, it will throw the exception we caught but you will lose all the information on the original exception stack.

## Common Exceptions

These are the most common exception types that you can throw or inherit from:

* System.Exception

The exceptions base class. Contain the following important fields:

* + StackTrace – all the function calls from the time the exception occurred
  + Message – a string that describe the error
  + InnerException – the internal error that caused this exception. Can contain additional exception(s)
* ArgumentException
* ArgumentNullException
* ArgumentOutOfRangeException
* InvalidOperationException
* NotSupportedException
* NotImplementedException
* NullReferenceException
* IndexOutOfRangeException
* IO.IOException
* Net.WebException - commonly thrown around any errors performing HTTP calls
* System.Data.SqlClient.SqlException – Various types of SQL Server exceptions
* System.StackOverflowException – If a method calls itself recursively, you may get this exception
* System.OutOfMemoryException – If your app runs out of memory
* System.InvalidCastException – If you try to cast an object to a type that it can’t be cast to
* System.ObjectDisposedException – Trying to use an object that has already been disposed

## Creating New Exception Types

Creating your own C# custom exceptions is really only helpful if you are:

* Going to catch that specific type of exception and **handle it differently**.
* Use it to track a very specific type of exception and/or to **monitor your application errors and logs for it** with an [error monitoring](https://stackify.com/error-monitoring/) tool.

## Debugging Exceptions

Sometimes, the exceptions messages are not very clear. In this case, we can add a try-catch block with the reported exception and this will allow us to run the debugger and catch the exception so that we can look into the returned exception object and find more meaningful details on it.

For example:

…  
try  
{  
 \_context.SaveChanges():  
}  
catch (DbEntityValidationException e)  
{  
 Cosnole.WriteLine(e); // we can break on this line and look into e   
}

### Finding First Chance Exceptions

Buggy code can throw many exceptions without handling them properly so you might see degraded performance without knowing why. For example:

public DateTime? GetDate(SqlDataReader reader, string columnName)

{

DateTime? value = null;

try

{

value = DateTime.Parse(reader[columnName].ToString());

}

catch

{

}

return value;

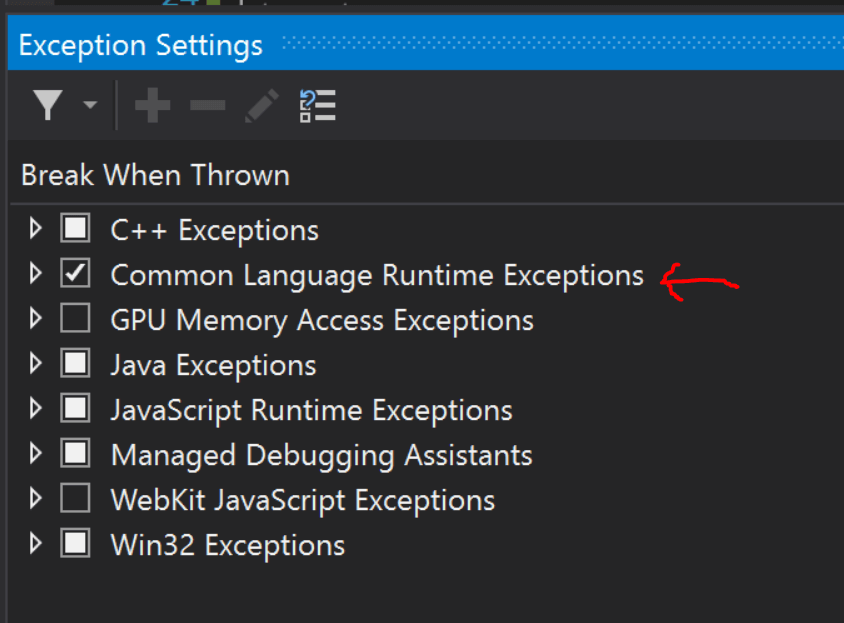
}

If this code is repeatedly called with invalid reader, columnName etc, the method will catch the exception without notifying the upper levels.

To find these cases, When you run your application within Visual Studio, with the debugger running, you can set Visual Studio to break anytime a C# Exception is thrown. This can help you find exceptions in your code that you did not know existed.

To access Exception Settings, go to Debug -> Windows -> Exception Settings

Under “Common Language Runtime Exceptions” you can select the types of exceptions you want the debugger to break for automatically. I would suggest just toggling the checkbox for all. Once you break on an exception, you can then tell it to ignore that particular type of exception to exclude it, if you would like.



## TryXXX Method Pattern

when appropriate, you can define pairs of methods:

* Tryxxxx - will try to perform the operation and will return an error code if failed (no exception) and
* xxxx - call Tryxxxx and if returns an error code - throws an exception.

This allows the user to choose the most appropriate path - either throwing an exception and looking at the returned value in some way.

## Exception Handling Best Practices

* <https://docs.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions>
* <https://kumarashwinhubert.com/10-exception-handling-best-practices-in-csharp>
* <https://stackify.com/csharp-exception-handling-best-practices/>

# Logging

## Windows Logging

* Console.WriteLine(…)
* Windows also support another logging target called **Debug Output**, which you can log to with System.Diagnostics.Trace("My message").
* You can log to the **Event Viewer**. This is a good but costly option (performance), so use only for critical errors.
* dotnet-trace - cross platform performance analysis utility - <https://docs.microsoft.com/en-us/dotnet/core/diagnostics/dotnet-trace>

## NLog

* popular logging framework for logging on different targets.
* <https://nlog-project.org/>

Setup: <https://github.com/NLog/NLog/wiki/Getting-started-with-ASP.NET-Core-2>

# ASP.Net

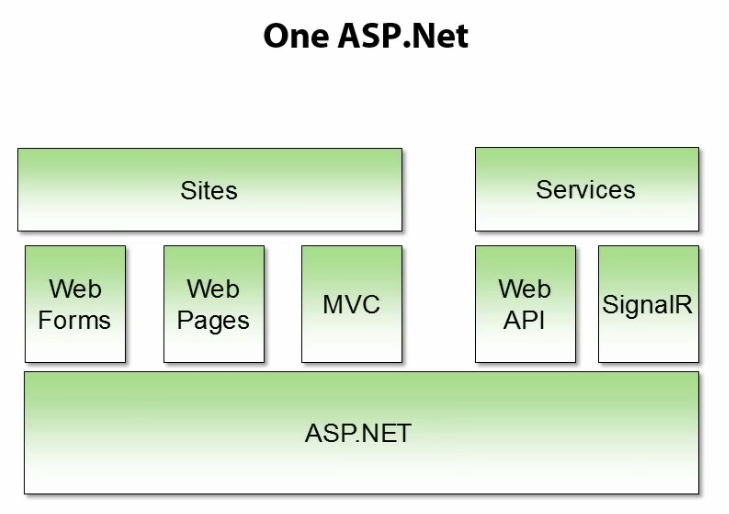
## ASP.Net

* ASP.Net
* ASP.Net MVC 2008
* Released to open source
* New syntax: Razon
* ASP.Net Web API with RESTful web services
* SignalR

## One APS.Net

Contains:

* Web Forms
* Model View Controller (MVC)
* Web Pages
* Web API
* SignalR



# C# Coding Standards and Naming Conventions

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions>

Below are our **C# coding standards**, naming conventions, and best practices.  
Use these in your own projects and/or adjust these to your own needs.

do: use **PascalCasing** for class names and method names.

1. public class ClientActivity
2. {
3. public void ClearStatistics()
4. {
5. //...
6. }
7. public void CalculateStatistics()
8. {
9. //...
10. }
11. }

**Why**: consistent with the Microsoft's .NET Framework and easy to read.

do: use **camelCasing** for method arguments and local variables.

1. public class UserLog
2. {
3. public void Add(LogEvent logEvent)
4. {
5. int itemCount = logEvent.Items.Count;
6. // ...
7. }
8. }

**Why**: consistent with the Microsoft's .NET Framework and easy to read.

do not: use **Hungarian** notation or any other type identification in identifiers

1. // Correct
2. int counter;
3. string name;
5. // Avoid
6. int iCounter;
7. string strName;

**Why**: consistent with the Microsoft's .NET Framework and Visual Studio IDE makes determining types very easy (via tooltips). In general you want to avoid type indicators in any identifier.

do not: use **Screaming Caps** for constants or readonly variables

1. // Correct
2. public static const string ShippingType = "DropShip";
4. // Avoid
5. public static const string SHIPPINGTYPE = "DropShip";

**Why**: consistent with the Microsoft's .NET Framework. Caps grap too much attention.

avoid: using **Abbreviations**. Exceptions: abbreviations commonly used as names,   
                 such as **Id, Xml, Ftp, Uri**

1. // Correct
2. UserGroup userGroup;
3. Assignment employeeAssignment;
5. // Avoid
6. UserGroup usrGrp;
7. Assignment empAssignment;
9. // Exceptions
10. CustomerId customerId;
11. XmlDocument xmlDocument;
12. FtpHelper ftpHelper;
13. UriPart uriPart;

**Why**: consistent with the Microsoft's .NET Framework and prevents inconsistent abbreviations.

do: use **PascalCasing** for abbreviations 3 characters or more (2 chars are both uppercase)

1. HtmlHelper htmlHelper;
2. FtpTransfer ftpTransfer;
3. UIControl uiControl;

**Why**: consistent with the Microsoft's .NET Framework. Caps would grap visually too much attention.

do not: use **Underscores** in identifiers. Exception: you can prefix private static variables   
                    with an underscore.

1. // Correct
2. public DateTime clientAppointment;
3. public TimeSpan timeLeft;
5. // Avoid
6. public DateTime client\_Appointment;
7. public TimeSpan time\_Left;
9. // Exception
10. private DateTime \_registrationDate;

**Why**: consistent with the Microsoft's .NET Framework and makes code more natural to read (without 'slur'). Also avoids underline stress (inability to see underline).

do: use **predefined type names** instead of system type names like Int16, Single, UInt64, etc

1. // Correct
2. string firstName;
3. int lastIndex;
4. bool isSaved;
6. // Avoid
7. String firstName;
8. Int32 lastIndex;
9. Boolean isSaved;

**Why**: consistent with the Microsoft's .NET Framework and makes code more natural to read.

do: use implicit type **var** for local variable declarations. Exception: primitive types (int, string, double, etc) use predefined names.

1. var stream = File.Create(path);
2. var customers = new Dictionary();
4. // Exceptions
5. int index = 100;
6. string timeSheet;
7. bool isCompleted;

**Why**: removes clutter, particularly with complex generic types. Type is easily detected with Visual Studio tooltips.

do: use noun or noun phrases to name a class.

1. public class Employee
2. {
3. }
4. public class BusinessLocation
5. {
6. }
7. public class DocumentCollection
8. {
9. }

**Why**: consistent with the Microsoft's .NET Framework and easy to remember.

do: prefix interfaces with the letter **I**.  Interface names are noun (phrases) or adjectives.

1. public interface IShape
2. {
3. }
4. public interface IShapeCollection
5. {
6. }
7. public interface IGroupable
8. {
9. }

**Why**: consistent with the Microsoft's .NET Framework.

do: name source files according to their main classes. Exception: file names with partial classes reflect their source or purpose, e.g. designer, generated, etc.

1. // Located in Task.cs
2. public partial class Task
3. {
4. //...
5. }
6. // Located in Task.generated.cs
7. public partial class Task
8. {
9. //...
10. }

**Why**: consistent with the Microsoft practices. Files are alphabetically sorted and partial classes remain adjacent.

do: organize namespaces with a clearly defined structure

1. // Examples
2. namespace Company.Product.Module.SubModule
3. namespace Product.Module.Component
4. namespace Product.Layer.Module.Group

**Why**: consistent with the Microsoft's .NET Framework. Maintains good organization of your code base.

do: vertically align curly brackets.

1. // Correct
2. class Program
3. {
4. static void Main(string[] args)
5. {
6. }
7. }

**Why**: Microsoft has a different standard, but developers have overwhelmingly preferred vertically aligned brackets.

do: declare all member variables at the top of a class, with static variables at the very top.

1. // Correct
2. public class Account
3. {
4. public static string BankName;
5. public static decimal Reserves;
7. public string Number {get; set;}
8. public DateTime DateOpened {get; set;}
9. public DateTime DateClosed {get; set;}
10. public decimal Balance {get; set;}
12. // Constructor
13. public Account()
14. {
15. // ...
16. }
17. }

**Why**: generally accepted practice that prevents the need to hunt for variable declarations.

do: use singular names for enums. Exception: bit field enums.

1. // Correct
2. public enum Color
3. {
4. Red,
5. Green,
6. Blue,
7. Yellow,
8. Magenta,
9. Cyan
10. }
12. // Exception
13. [Flags]
14. public enum Dockings
15. {
16. None = 0,
17. Top = 1,
18. Right = 2,
19. Bottom = 4,
20. Left = 8
21. }

**Why**: consistent with the Microsoft's .NET Framework and makes the code more natural to read. Plural flags because enum can hold multiple values (using bitwise 'OR').

do not explicitly specify a type of an enum or values of enums (except bit fields)

1. // Don't
2. public enum Direction : long
3. {
4. North = 1,
5. East = 2,
6. South = 3,
7. West = 4
8. }
10. // Correct
11. public enum Direction
12. {
13. North,
14. East,
15. South,
16. West
17. }

**Why**: can create confusion when relying on actual types and values.

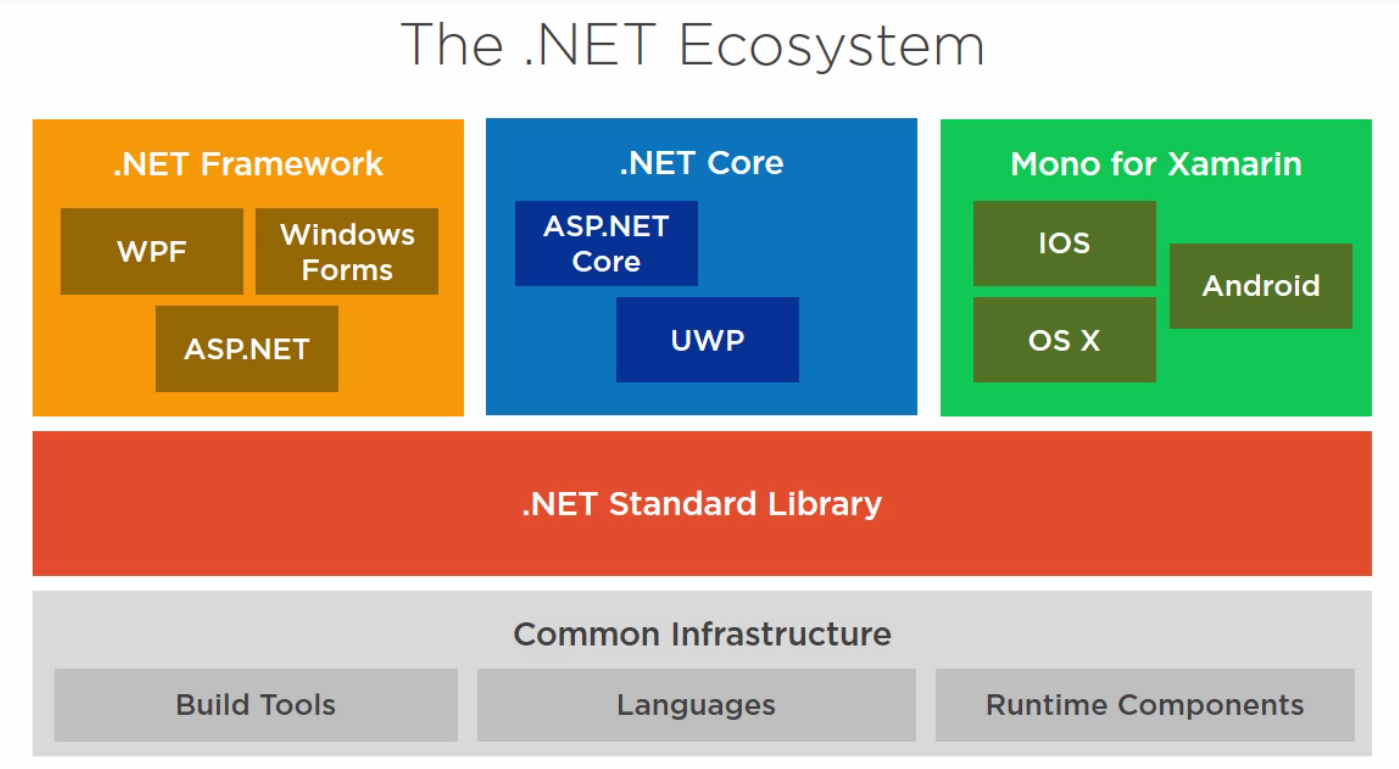
do not suffix enum names with Enum

1. // Don't
2. public enum CoinEnum
3. {
4. Penny,
5. Nickel,
6. Dime,
7. Quarter,
8. Dollar
9. }
11. // Correct
12. public enum Coin
13. {
14. Penny,
15. Nickel,
16. Dime,
17. Quarter,
18. Dollar
19. }

**Why**: consistent with the Microsoft's .NET Framework and consistent with prior rule of no type indicators in identifiers.

# .Net Ecosystem

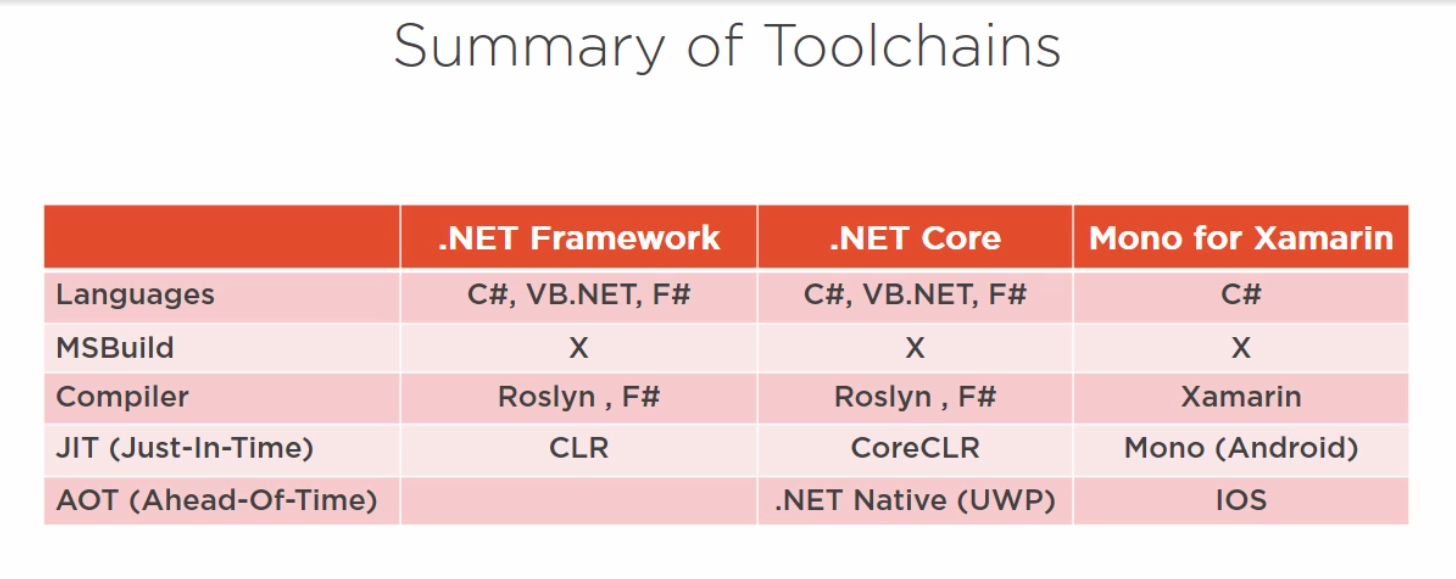
The .Net Ecosystem have a number of runtimes:



* Runtime frameworks:
  + .Net Framework – windows specific with support for windows API
  + .NET Core – multi-platform with ability to run side by side, better performance
  + Mono – for Android and iOS development
* .Net Standard Library – a specification of .Net API with interfaces for all the runtimes.
  + .Net Standard is a specification and not something physical that you can install!
  + It contains the formal specification of the .Net APIs
  + It’s the evolution of the Portable Class Libraries (PCL)
  + The different runtimes (.Net Framework, .Net Core and Mono) implement the .Net Standard.
  + Each runtime version implements a specific .Net Standard version.
  + The purpose of the .Net Standard is to allow sharing code between the different runtimes.
  + .Net Standard is backward compatible. We can rely that future .Net Standard version will not break the current version.
  + Each new version contains additional APIs but it doesn’t change the existing APIS.
  + Since Higher versions means more APIs and lower versions means that the version is supported on more platforms,   
    You should always target the lowest version you can.
* Common Infrastructure – include all the .Net languages, compiler and CLR, garbage collection, JIT etc.

### Comparison

|  |  |  |
| --- | --- | --- |
| .Net Framework | .Net Core | Mono for Xamarin |
| Partially open source (can view part of the code but not change it) | Completely open source | Completely open source |
| Only on Windows | Cross platform: Windows client, server, IoT, Linux, Mac | Cross platform: Android, OS X, IOS |
| C#, F#, VB .Net | C#, F#, VB .Net | C# only |
| Console applications WPF, ASP.Net Azure (WebJobs, Cloud Services) | Console applications APS.Net Universal Windows Apps (UWP) | Apple, android applications |
| Need to install .Net Framework to run .Net applications. | Need to install .Net Core runtime to run |  |
| .Net runtime framework includes the CLR and Class Library. | .Net Core includes the Core CLR, Core Class Library and the App Host that runs them |  |
| Major .Net versions can run side by side but not minor versions. | All .Net Core can run side by side. | No need/option to install the run time. |
| Does not support self-contained application. | You can also run self-contained application that contains the .Net Core within it and doesn’t require the user to install it to run. | All applications are self-contain applications. You never have to install Mono runtime. |



## .NET Framework

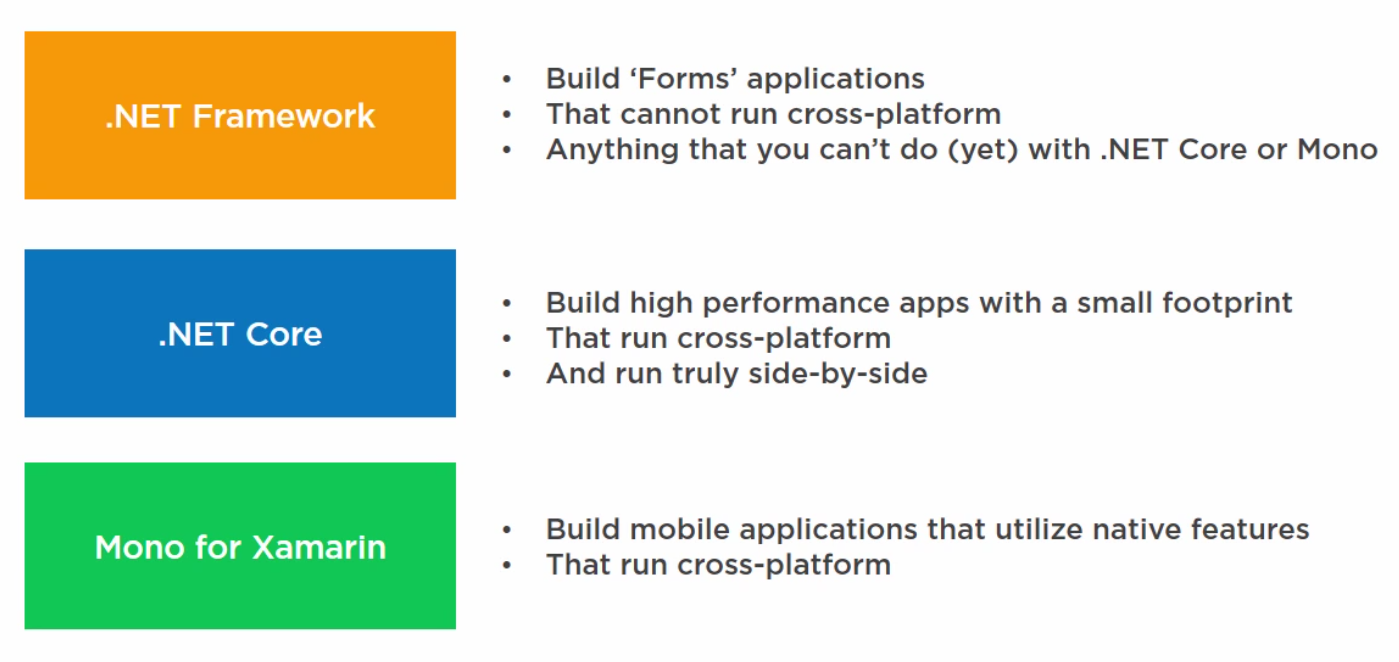
### Toolchain

## .Net Core

## Mono for Xamarin

## What to Use

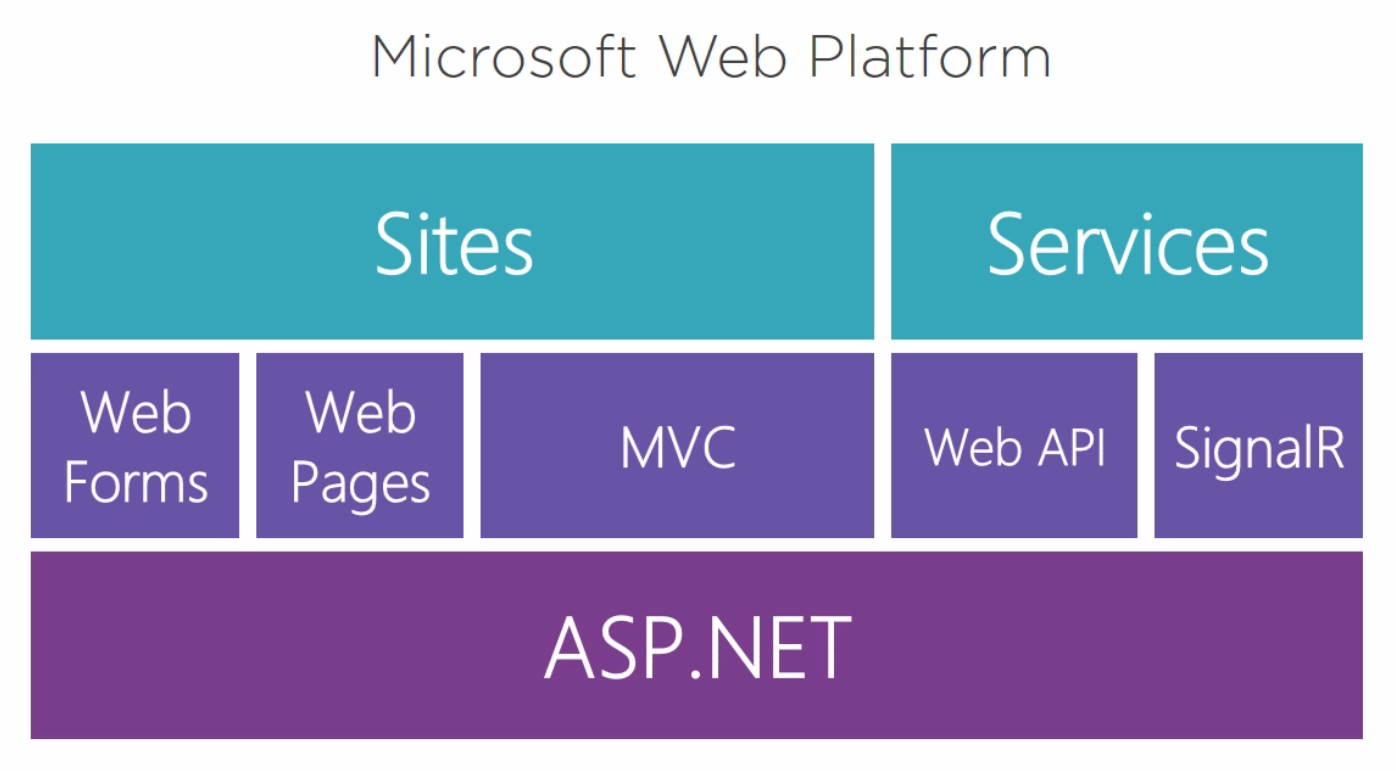
* **Any application that can be developed on .Net Core, should be developed on .Net Core**. Example: APS .Net and cross platform applications. In addition for it being cross platform, it also have a small footprint and has much better performance than the same application on .Net Framework.
  + Choose Framework-dependent app if you want it to be independent of the target OS and if you want the application itself to be small.   
    However, the negatives are that .Net Core will need to be installed on the system and that upgrading the .Net Core on the target might break the app.
  + Self-contained apps don’t require installing the .Net Core runtime and will not break with future .Net Core runtime versions. However, it’s much larger than the framework-dependent application and it targets only one OS at a time.
* Any application for mobile, should be developed on **Mono**
* Any other application that is windows only, should be developed on .Net Framework.
* If you want to share some code between the different types of runtimes (.Net Framework, .Net Core and Mono), like if you’re writing Nuggets, you can write it as a .Net Standard Shared Class Library (choose the lowest version that meets your need) and this class library can be used by the other runtime applications.



# Web Development with ASP.NET

## General

* + ASP.Net life-cycle events and stages
  + ASP.Net vs MVC
    - ASP.Net is a web development framework
    - It has services to build web applications
    - It’s part of the .Net framework with access to .Net libraries.
    - MVC – Model-View-Control design pattern. It makes developing, testing and maintenance easier.



* + WebAPI – to invoke functions over http
  + SignalR – for realtime communication
  + Web Forms – drag&drop event-based framework for web development. Not supported in ASP.Net Core
  + Web Pages - Mix the server-side code into the markup (html)
  + MVC – Mature MVC framework for developing websites.

### State Management

* + Session State:  
    Data is saved for a single user on a single browser. Saved on the server side.  
    Note: session state kills scalability of web application because the more users we have, the more data we need to save on the server.   
    In modern web app, we move toward cloud-based and stateless applications. So that when a request comes in, we server the response and not save any additional user-specific data on the server.
  + Application State:  
    Allows multiple users to share data. Saved on the server side.
  + Cookies:  
    A single user on a single browser. Data is saved on the client side.
  + View State:  
    Data is saved on the client side as viewstate. Classes must be marked serializable and must be cast on the way out.

### Data Validation

* + On client side:
    - Quicker and responsive to the user
    - Easily spoofed (hacked or worked around by the user)
    - Unobtrusive Client Side Validation:
      * Uses ASP.Net
      * Set in Global.asax
      * Add jQuery
      * Add AspNet.ScriptManager.jQuery
  + On Server Side:
    - Resistant to spoofing and therefore more reliable
    - Requires a postback to the servers and therefore, slower.
  + The best approach is to use both.

### ASP.Net vs ASP.NET Core Comparison

|  |  |
| --- | --- |
| ASP.NET | ASP.NET Core  Complete re-thinking of the entire .Net platform! |
| Requires windows | Cross platform |
| Web Forms, Web Pages, MVC | Web Pages, MVC |
| Requires the full .NET Framework | Can work with the full .NET (.Net 4.x) Framework or with the lightweight .NET Core |
|  | Unified syntax and controllers |
|  | Dependency injection out of the box. |
|  | Single platform for MVC and WebAPI |
|  | Multiple deployment support: cloud, IIS, self hosting (on windows with cli) etc. |
|  | Low memory footprint |
|  | Completely composed: everything above the .Net Core Layer is just packages that we can install through Nuget. Including MVC, Logging, Identity etc.  This means that everything is optional |

## Razor

Razor is a template markup syntax that lets you embed server code in an HTML page.

Razor syntax is based on the ASP.NET framework, the part of the Microsoft.NET Framework that's specifically designed for creating web applications.

Razor web pages can be described as HTML pages with two kinds of content: HTML content and Razor code (either C# or VB code).

When the server reads the page, it runs the Razor code first, before it sends the HTML page to the browser. The code that is executed on the server can perform tasks that cannot be done in the browser, for example accessing a server database. Server code can create dynamic HTML content on the fly, before it is sent to the browser. Seen from the browser, the HTML generated by server code is no different than static HTML content.

The Razor syntax is used in Web Forms and MVC. It is much cleaner than the original WebForm syntax.

### Razor with C#:

* Razor code blocks are enclosed in @{ ... }
* Razor expression starts with @myVar.Value if the expression is not clear, you can add parenthesis:   
  <img src=”[~/img/@(p.ArtId).jpg](mailto:~/img/@(p.ArtId).jpg)”>
* The razor syntax also works inside html strings like the example above and like:  
  <…. id=**”@**Model.Category.Id**”**…>
* Inline expressions (variables and functions) start with @
* Code statements end with semicolon
* Variables are declared with the var keyword
* Strings are enclosed with quotation marks
* Use @: or <text></<text> to display text from code block.
* C# code is case sensitive
* C# files have the extension .cshtml
* **Comments:**

**@\*** this is a multiple  
 Line comment!  
 **\*@**

### Dynamic Rendering

**@{**  
 var popularClassName = Model.Customers.Count > 5 ?  
 “popular” **:   
 null**;  
**}  
…**  
<h2 class=**”@**popularClassName**”**>…</h2>

### User Input

@{  
var totalMessage = "";  
if(**IsPost**)  
    {  
    var num1 = Request[**"text1"**];  
    var num2 = Request["text2"];  
    var total = num1.AsInt() + num2.AsInt();  
    totalMessage = "Total = " + total;  
    }  
}  
<html>  
<body style="background-color: beige; font-family: Verdana, Arial;">  
<form action="" method="**post**">

<p>  
 <label for="text1">First Number:</label><br>  
 <**input** type="text" **name="text1"** />  
 </p>  
 <p>

<label for="text2">Second Number:</label><br>

<input type="text" name="text2" />

</p>  
<p>

<input type="submit" value=" Add " />

</p>  
</form>  
<p>@totalMessage</p>  
</body>  
</html>

* IsPost – will return true if the form was posted (see form action above)
* Request[“<name of the html input element>”] – return the content of the html input element with this name.

### Rendering

Razor will render parts of the page dynamically using the @Render commands:

* @RenderBody() – will render the content of the page’s body. This is used in Shared/\_Layout.cshtml to render the main contents of the page that specified \_Layout as its ‘Layout’.
* **@RenderSection(“<section name>”,<does this section exist on every page?>)** – will render the section with the matching name in the page.  
  For example:  
  @RenderSection(“scripts”,false)  
  will render   
  @section Scripts {  
   ….  
  }  
  From the page.

## Web Forms

New->Project->Web->ASP.NET Web Application

Choose ‘Empty’ an Add folders: Web Forms , Add unit tests



* + Object oriented
  + Event driver
  + Supports separation of markup from logic
  + Pre-compilation or dynamic compilation
  + Remarkable integration with Visual Studio

|  |  |
| --- | --- |
| Web Forms Strengths | Web Forms Challenges |
| Established maturity | Bloated:  \* Object model and events \* Viewstate |
| Rapid application development.  Can be used by beginner developers | Not supported in ASP.NET Core |
| Strong 3rd party support |  |

### Technical

* + Use HTML Contols
    - Add runat =”server” to any control that we want to control in code.
    - Supports only two HTML events:
    - onServerClick
    - OnServerChange
  + Support Web Controls which allow richer controls composed of HTML controls

### Controls Notes

* + RadioButton – need to have the same ‘GroupName’ to be mutually exclusive
  + Validation Controls:

In order to use Validation Controls:

1. Set in Global.asax

In Global.Application\_Start add:

ValidationSetting.UnobtrusiveValidationMode = UnobtrusiveValidationMode.WebForms;

1. Add jQuery

Tools->Library Package Manager->Manage NuGet Packages for Solution… or Package Manager Console

>> install-package jQuery

1. Add AspNet.ScriptManager.jQuery

>> install-package AspNet.ScriptManager.jQuery

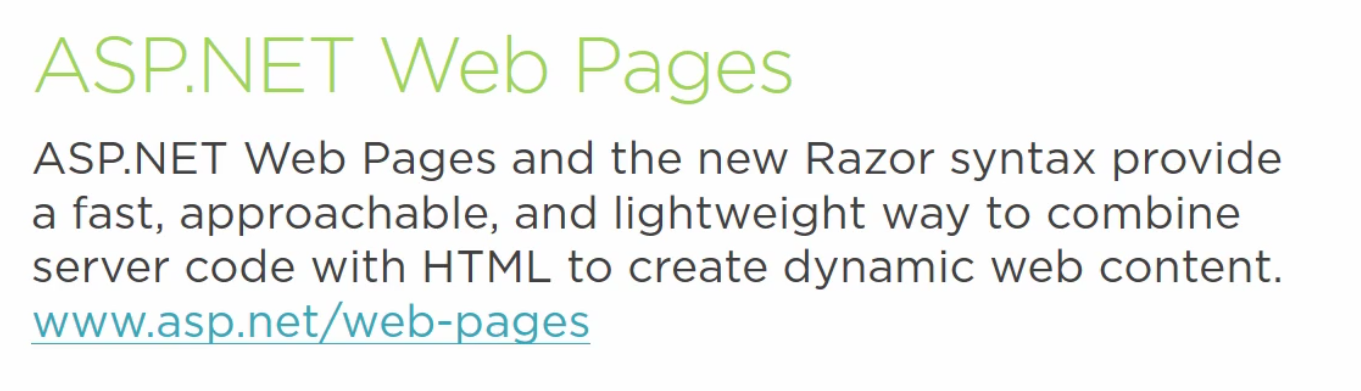
* + Types of Validation Controls:
    - RequiredFieldValidator
    - CompareValidator
    - RangeValidator
    - RegularExpressionValidator
    - CustomValidator
    - DynamicValidator
    - ValidationSummary
  + Data Sources – Connect a data source to your application:
    - SqlDataSource  
      Configure Data Source-> New Connection… -> Microsoft SQL Server
    - LinqDataSource
    - ObjectDataSource
    - XMLDataSource
    - SiteMapDataSource
    - AccessDataSource
    - EntityDataSource
  + Data Control – a control for displaying the data in the web form:
    - Chart
    - DataList
    - DataPager
    - DetailsView
    - FormView
    - GridView – very popular!

Connect Data source:

* + In a new form, choose Data->SqlDataSource->New Connection. Enter the SQL server and the data file.

## .Net Web Pages

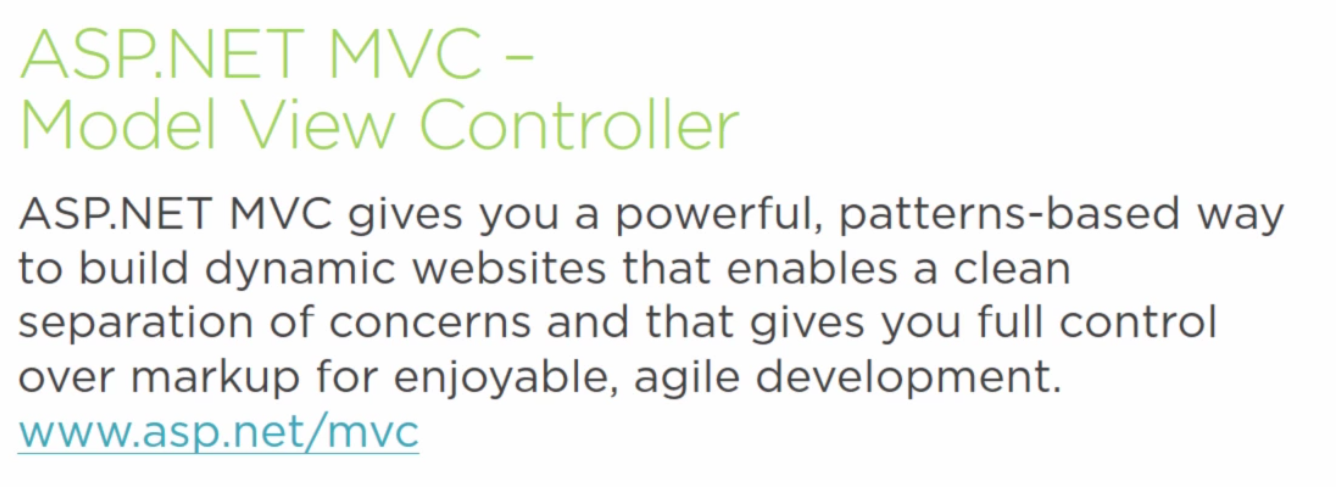
New->Web Sites… -> ASP.NET Web Site (Razon v3)



* + Mix the server-side code into the markup (html)
  + Uses Razor syntax to execute all the code in the web pages
  + Dynamic compilation – when the page is 1st requested on the server

|  |  |
| --- | --- |
| Web Pages Strengths | Web Pages Challenges |
| Server code is embedded in HTML Convenient for experienced web developer | Server code is embedded in HTML Does not follow separation of concerns and is not easily testable |
| Dynamic compilation Can deploy just the one file that changed | Smaller adoption |
| Supported in ASP.NET Core |  |

## MVC Model-View-Controller



* + Emphasis on “separation of concerns”
    - Model – data
    - View – user interface (html)
    - Controller – processing.
      * The controller tells the view what to display and handles requests from the html.
    - Router – always exist in an MVC app – responsible for receiving the http request and selecting the controller to handle it.
  + Mature and continuously developed by Microsoft

|  |  |
| --- | --- |
| MVC Strengths | MVC Challenges |
| Emphasis on ‘Separation of concerns’ | Requires stronger developer skills |
| Supported in ASP.NET Core |  |
| Strong adoption |  |

### Architecture

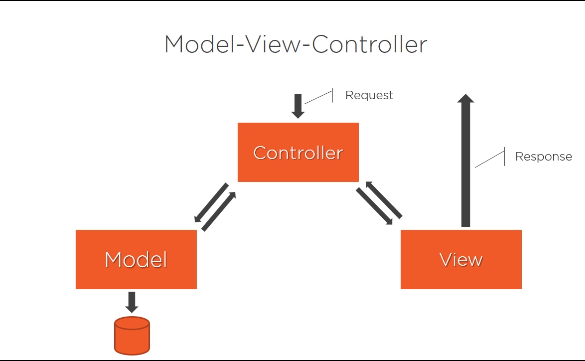
#### Client Side

The client sends an http request to the server

#### Server Side

On the server:

* + The router receives the request and calls the appropriate controller to handle it.
  + The controller handles the request (action) and send the data to return to the view
  + The view, with the help of the Razor engine, creates the full static HTML page and send it back to the client.



### Convention Over Configuration

* + Controllers naming: Product**Controller**
  + Controllers live in folder Controllers:  
    Controllers/ProductController.cs
  + Views live in folder Views
  + Views named according to the controller:
    - Views for ProductController will live in Views/Product
  + Each view named for an Action in the controller

Views/Product/Edit.cshtml

* + Convention over Configuration in Entity Framework – Code First.

### MVC Project

#### Build Configurations

1. In the Solutions Configurations -> Configuration Manager
2. Under Project-Configuration->New.
   1. Give Name and copy initial setting from existing configuration.

**Recommendation**: Work with Debug configuration ONLY when you’re developing. For everything else, including testing – **use** **Release** (and derivative) **configurations**!

1. In solution explorer->Web.config->right click->Add Config Transform

This will create a new Web.config file for the new build.

#### Application Setting

Application setting such as email server address, user names and passwords should not be hard-coded in our code. Instead, they should be saved in the Web.config file for the relevant build.

In the Web.config file, **<appSettings>** contain all the application settings and where we can store all our application setting as key-value pairs.

In our application, we can access these values through:  
ConfigurationManager.AppSettings["<key>"] or  
ConfigurationManager.ConnectionSettings["<key>"]

This will return a string that we might need to convert to the required type in our code.

Example:  
In Web.config:  
<add key="FacebookAppId" value="12345" />

In code:  
ConfigurationManager.AppSettings["FacebookAppId"]

#### Build-Specific Application Setting

In order to override an application setting, we’ll add a <appSettings> tag to the Web.<build>.config file and for every setting that we want to override, we’ll add the new value and an xdt:Transform and xdt:Locator attributes:

<appSettings>  
 <add key="<name of the key we’re overriding>"   
 value="<the new value for the key>"  
 xdt:Transform="SetAttributes"   
 xdt:Locator="Match(key)"/>  
</appSettings>

Note: the Match(key) contain the type of settings we’ve added in this case, key because of the <add key=…>

#### Securing Application Setting

Problem:

The user names and passwords are all in plain text in the Web.config files so when they are committed to the repository, everyone can see them. This is not a problem if you’re working on your secure and private repository but might be an issue if you’re working on GitHub for example.

Solution:

For secret data that shouldn’t be part of the repository:

1. Solution explorer -> right click -> Add -> new item -> Web -> Web Configuration File
2. Move all the secret configurations to the new Configuration file (e.g. AppSettings.config)
3. In the original appSettings tag, add: file=”<new configuration file>”

For example:  
in Web.Config:

<appSettings **file**="AppSettings.config">  
 … all the settings that   
 can be committed to the repository  
</appSettings>

1. If you want to replace the whole tag (when all the settings are secret without any shared settings), you can replace the ‘file’ with ‘configSource’:

<appSettings **configSource**="AppSettings.config" />

NOTE: Make sure that you **don’t commit the secret configuration files** with the project!! (e.g. add them to .gitignore)

#### Encrypt Setting on Deployment

All our settings get deployed to the server in plain text. So we need to encrypt them on deployment:

1. Deploy the app to the file system as usual
2. In Windows Start -> Visual Studio Tools -> Developer Command Prompt -> Run as admin
3. Aspnet\_regiis -pef “<section to encrypt>” “<path to the deploy folder>” -prov “<encryption/decryption provider>”  
   this will encrypt the requested section.  
   For example:  
   aspnet\_regiis -pef “appSettings” “c:\deploy” -prov “RSA”

\*\* There are more details to this if/when I need it, I’ll have to learn it better.

Note: in real-life application, you will use a deployment automation process that will include this encryption.

### MVC Application Structure

#### App\_Start

**RouteConfig.cs** – contains our router configuration that determine which controller and function to call for each URL path request.

**BundleConfig.cs** – containsthe definitions of the different external css and js files that can be used in the different pages.

**FilterConfig.cs** – define global filters like the ‘HandleErrorAttribute’ for handling application exceptions or the ‘AuthorizeAttribute’ for requiring authorization on all the application

#### Content

Where we store all our client-side assets such as picture, audio, bootstrap etc.

#### Controllers

Where all our controllers live

By default, will have the HomeController

#### Fonts

Where all our fonts are stored

#### Models

All our domain classes will be here.

#### Scripts

Java scripts

#### Views

All the views. See ‎9.4.1 Convention Over Configuration for naming conventions.

* + - \_ViewStart.cshtml – contain the default Layout for all views in the application. Everything in this file will be put in every other view file at the beginning.
    - Shared\\_Layout.cshtml – the main html layout of the site. Includes the head and body tags etc.

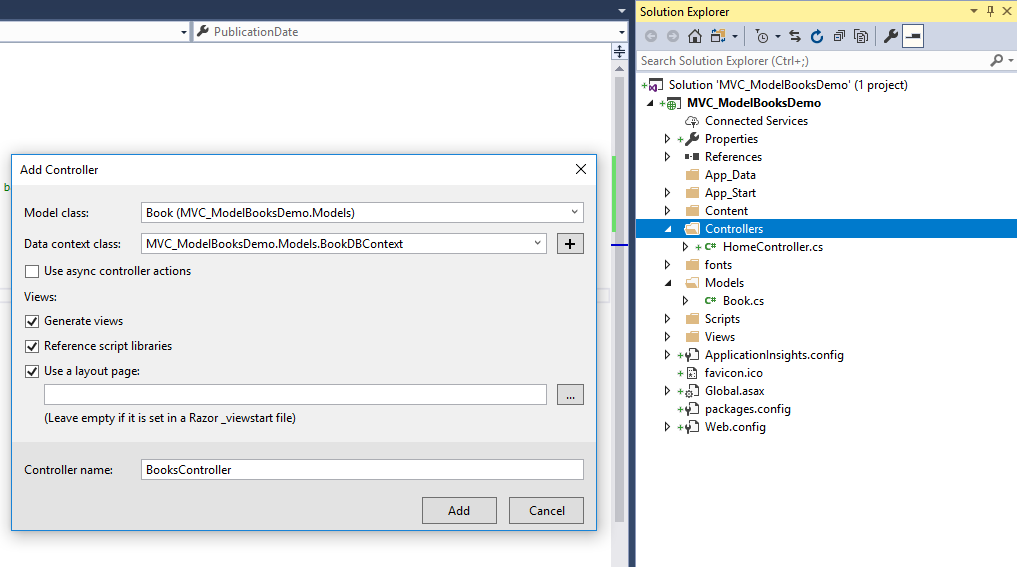
#### Additional Files in Root

* + **Packages.config** – all the packages (and their versions) that are used in this app.
  + **Global.asax –** a class the provide hooks for various events in the application life cycle.

### Controllers

Controllers->Add->New Scaffold Item->MVC 5 Controller with views, using Entity Framework

* + Do not directly control the HTML from the controller (although you can).
  + Use HTMLEncode helper when displaying data from user. This will protect our site from attack
  + Every method in the controller class is an action that we can navigate to from our URL.



* + Every action in the controller returns ActionResult or a derivative of the ActionResult class.
    - Good practice: If we return View or ViewResult, we should return ViewResult in the action method definition. For example:  
      public **ViewResult** GetMovie(int id){  
       …  
       return View(movie);  
      }
  + Additional supported **ActionResult** types. Note: they all inherit from ActionResult abstract bas class:

|  |  |
| --- | --- |
| Returned Type | Helper Method |
| ViewResult | View() |
| PartialViewResult | PartialView() |
| ContentResult | Content(string to display) |
| RedirectResult | Redirect |
| RedirectToRouteResult | RedirectToAction(“<action>”,”controller”, anonymous arguments for action) RedirectToAction(“Index”, “Home”, new {page = 1,   sortBy = “name” }); |
| JsonResult | Json() |
| FileResult | File() |
| HttpNotFoundResult | HttpNotFound() |
| EmptyResult | return new EmptyResult(): |

#### Actions Parameters

The MVC framework map any parameters data to the controller’s action method parameters. The source of the parameters data can be from:

* + - URL: /movies/edit/1

Note: the parameter that the route will be mapped into is defined in RouteConfig.cs and is by default (and in this case) {id}

* + - Query string: /movies/edit?id=1
    - Form data: id=1

#### Optional Parameters

To add optional parameters, we need to add them to the method’s definition. Note: they must be nullable so that if the user doesn’t send anything, the action method will perform some default action:  
public ActionResult Index(int? pageIndex, string sortBy)

#### Accessing Query String

In the controller action:

* Check if a certain query string exist (e.g. ReturnUrl):  
  if( Request.Query.Keys.Contains("ReturnUrl") )
* Use the value of the URL string:  
  Request.Query["ReturnUrl"].First()

#### Convention-Based Custom Routes – For MVC < 5

In order to avoid having to write all the optional parameters in the URL query every time, we can add them as custom routes to RouteConfig.cs

Note: we have to add them before the default route since the routes processing is done from the more specific to the more general one.

#### Attributes-Based Custom Routes – For MVC 5 and Up

In MVC 5, MS added support for a much better way to add custom routes:

* + 1. In RouteConfig.cs enable attribute routes by:  
       routes.MapMvcAttributeRoutes();
    2. In any controller’s action method that works with custom routes, we add the route in a [Route(…)] attribute in the method’s definition:

[Route("movies/releases/{year:regex(\\d{4}):range(1900,2019)}/{month:regex(\\d{1-2}):range(1,12)}")]  
public ActionResult ByReleaseDate(int year, int month)

The main supported constraints on the route are:

* + - * regex
      * range
      * min/minlength and max/maxlength
      * int/float/guid
      * etc

#### Attribute: HttpGet (default) vs HttpPost

Every action in the controller will be considered as a Get action, unless it is annotated by the **[HttpPost]** annotation.

Model Binding: MVC is smart enough to bind the @model or sub-class of the @model to our Post Action method

[HttpGet(“<path>”)] – for setting a custom-path for specific actions instead of the default {controller}/{action}/{id?} path.

#### Passing Data to the View

We can use one of the following ways to pass data to the View from the controller:

* + - * **ViewData**: ViewData is a dictionary which can contain key-value pairs where each key must be string. It can be used in the controller’s action method to pass data to the view.  
        **Note:** ViewData only transfers data from controller to view, not vice-versa. It is valid only during the current request.  
        This method is very ugly (unreadable code) and not type-safe and therefore, not recommended!!
      * **ViewBag**: Use this when you don’t want to return the full model but instead return partial, calculated or temporary data instead.  
        ViewBag=ViewData + Dynamic wrapper around the ViewData dictionary. Like the ViewData, this is not the best method to use unless you’re moving data from one view to another.
        + **Can be used for adding fields on the fly that can be accessed in the view**.   
          For example:  
          in the Controller:  
          ViewBag.SomeNewData = “This is my new data”;

And in the matching view:

<p>@ViewBag.SomeNewData</p?

As long as ViewBag.SomeNewData is not used – this will be an empty tag that won’t show anything. Once we set its value, it will be shown on the page.

* + - * The most recommended method: using ViewModel!!
        + If you need to return the whole model as defined in /Models, just return it in the action method:  
          return View(<model object to display>)
        + If you need to return custom/calculated/temporary data:

Add a folder /ViewModels under the main project

Add a new (poco) class: {Controller}{View}ViewModel for example:  
MoviesRandomViewModel and define all required fields in it.

In the cotroller action-method, create a new ViewModel object, initialize it and send it to the View:  
var moviesRandom =   
 new MoviesRandomViewModel(){…};  
return View(moviesRandom);

In the View, add the @model line to recognize the Model as the passed class:  
@model AspDotNetCourseApp.ViewModels.MoviesRandomViewModel

#### Unit Tests

Unit test project name: <project under test>.ControllerTests

Things to test: That the controller returns the expected:

* + 1. View
    2. HTTP status code
    3. Redirect to URL
    4. Correct model for the view

### Views

Views->Add Folder with the controller’s name

New Folder->Add->New Scaffolded Item->MVC 5 View – Empty

And name it to the Action in the controller

Note: If you check ‘Create as a partial view’ – it will create a kind of widget that you can re-use in different views (React.Component?).

Choose ‘Use a layout page’ to Views\Shared\\_Layout.cshtml – set the current name everywhere.

* + Controllers should not know about the display – that’s the views job
  + Controllers pass data to the View
  + Views use the Razor syntax (.cshtml files).

#### Partial Views

Convention: their name should start with an underscore. For example:  
Views\Shared\\_NavBar.cshtml

**To render a partial view:**  
@Html.Partial(“<name of partial view>”,<optional model for view>)

* To pass an optional model view, it needs to be defined at the top of the current view (with @model line). Or derived from the @model line.
* If we don’t include an optional model for the partial view, the @model defined at the start of the page, will be passed on automatically.

#### IEnumerable

When we’re just displaying the data in a collection, we should try to use IEnumerable<T> as the interface instead of the actual collection type in order to de-couple the code to the concrete collection type in the model.

#### HTML Helpers

Used for rendering HTML controls in the view.

##### Forms

**@using Html.BeginForm(<action>,<controller>){}**

* Will render <form>… and in the closing of the using block, it will call the .Dispose method that will render the form closing tag (</form>
* On “Submit” will call the <action> method in the <controller> and will pass the submitted data as the @model. This is called **Model Binding** in MVC. Note: MVC is smart enough to perform the binding also on a member of @model if all the data in the form belong to this member.

##### Button

<input type="button" onclick="location.href='@Url.Action<Action>, <Controller>)'" />

Where <Action> is the Action Method inside the <Controller>

For example:  
<button class="btn btn-outline-primary" onclick="location.href='@Url.Action("New", "Customers")'">Add New Customer</button>

##### Inputs:

* + @Html.Label
  + @Html.DropDownList
* Strongly Typed Inputs:

Will perform validation (according to the type and data annotations in the field’s definition in the model) and will take the text from the member name (unless annotated with Display()):

* + Html.LabelFor
  + Html.DropDownListFor
* Templated Helpers:

Note: the ‘For’ variation that connects to model’s fields will perform automatic validation of the field!

* + Html.TextBox(For)
  + HtmlEditorFor
  + Html.Hidden(For)  
    When you need to pass data to the action Handler function but you don’t want to show it on the form, you can pass it through Html.HiddenFor function:  
    @Html.HiddenFor(m => m.Customer.Id)
  + Html.Password
  + Html.RadioButton(For)
  + Html.CheckBox(For)
  + Html.ActionLink:

Html.ActionLink(

article.Title, // **Text to display**

"Login", // <-- **ActionMethod**

"Item", // <-- **Controller Name**

new { id = article.ArticleID }, // <-- **Route arguments**.

null // <-- **htmlArguments**

)

@using

### Model

Models->Add->Class

This will create a POCO (Plain Old CLR Object), meaning an empty class

* 1. Model represent data
  2. Model can be manipulated by the controller
  3. Controller can pass model to the view
  4. View can use model for type-safe display
  5. We can add validation/annotation with ‘Data Annotation’:  
     [Display(Name=”Display This Name”)]   
     For a full list of annotation, see   
     The data annotation (System.ComponentModel.DataAnnotations) provides both client and server validation checks with no additional coding required by you.  
     You can also create custom validation attributes.

### ViewModels

Especially when creating WebAPI where our external API needs to be stable, but no only, you can define a ViewModel class that holds only the properties that you need to pass to the View (or the API). And map between it and the model via an automatic mapping library (e.g. AutoMapper).

The benefits of using the model-viewModel-View patters are:

* To allow for your view to receive/post more/less information than what your domain model provides.  
  Even if they are the same on day one, they can (and usually do) diverge over time so it's easier to do the work up front, and follow good design, than to try to fix it later when you need it.
* **Decoupling** your view from your domain which can cause fragility if your domain evolves (see previous point).
* **Security**: the security issue is not in passing the model the view, but rather in receiving posts that are automatically bound to the data model. If you have sensitive information in there, it's possible for someone to craft a post that will change things you didn't intend.

See <http://odetocode.com/blogs/scott/archive/2012/03/12/complete-guide-to-mass-assignment-in-asp-net-mvc.aspx>

* A separate ViewModel **stops abstraction leaking** that may come along with the DB Model. This is true especially if you are using EF with navigation properties enabled.
* They aren't a necessity but they follow good design principle of abstraction and de-coupling.

### Database Models

Models that represent database tables must have:

**Id** or <model class>Id public property that identify a record in the table. For example:  
public class Customer   
{  
 public int Id {set; get;}  
}

#### Navigational Property

When one model is connected to another model, we can:

* Add a reference to the 2nd model:  
  …  
  public MembershipType MembershipType { get; set; }
* Or add just the 2nd model’s Id as a foreign key:  
  …  
  public byte MembershipTypeId {get; set; }  
  Entity framework know this convention and will treat any <modelClass>Id property as a foreign key to the <modelClass> model.

### Database Management

* 1. While developing/testing the application, we might need the ability to:
     1. Reset and Seed the database with starter values
        + Create a (BookDBInitializer) class derived from DropCreateDatabaseAlways<T>
        + Override the parent’s ‘Seed’ function. In it:
          1. Add new items to the BookDBContect.Books
          2. Call base seed method
        + In Global.asax.cs.Application\_Start():  
          Database.SetInitializer(new BookDBInitializer());
     2. Add new fields and recreate the controller & views automatically
        + To add a new field:
        + Add the field in the model
        + Recreate the same controller as before. Press ‘Yes’ to replace existing controller. NOTE: This will only work if you didn’t make any changes to the Visual Studio automatic code!

### Forms

**remember to treat any data in an HTTP request as malicious until proven otherwise.**

* 1. Forms have an action and a method
     1. Action – where to send the form. The default is to post the form back to itself.
     2. Method – Post/Get
  2. Use Get when you are not changing the object. Get should be idempotent (will not change when running it over and over again)
  3. Use Post when you are updating the object

### Data Annotations

using System.ComponentModel.DataAnnotations;

<https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations>

Using Data Annotations in the Model class allows us to:

* + Override the default data types that Entity Framework uses for specific fields in the database.
  + This will provide both client side and server-side validation for input fields.

Example:

using System.ComponentModel.DataAnnotations;  
…  
[Required]  
[StringLength(255)]  
public string Name { get; set; }

This will override the default data type of Name to not allow null and to limit its maximum size.

#### Commonly Used Data Annotations – Display + Validation

* + [Required]
  + [StringLength(num)]
  + [RegularExpression(…)]
  + [Range(..)]

#### Display-Only Data Annotations

The following data annotations are used for display only and are not used for validation:

* + [Display(Name=”Display This in HTML As Labe”)]  
    set the text to display in the html for this field.  
    The disadvantage of this approach is that whenever we change the label of this field, we’ll need to re-compile our code.   
    The advantage of this approach is that labels the fields automatically and therefore, if we rename the fields, the label will be updated automatically, unlike if we label the field in the html itself where the VS will not update it.
  + [DataType(DataType.<data type>)]  
    define a more specific data type to use for this field’s display. For example:  
    [DataType(DataType.EmailAddress)]  
    public string EmailAddress {set; get; }
  + [DisplayFormat(…)]

### AJAX

Allow creating a faster, smoother user interface by having all the request work asynchronously without freezing the user interface while it waits for the server’s response.

There are two options for using Ajax:

#### .Net Ajax Library

Used for calling actions in MVC controllers. Can also be used for calling Web API actions but then the syntax gets complex and ugly.

1. References->Nuget Packages management-> install Microsoft.jQuery.Unobtrusive.Ajax and jquery.validate.unobtrusive.js

Make sure that the following jquery scripts where added to your Scripts folder:

jquery.unobtrusive-ajax.(min.)js  
jquery.validate.unobtrusive(.min).js

1. In Views->Shared->\_Layout.cshtml: in the rendering section add:  
   @Scripts.Render("~/ bundles / jqueryval")
2. In App\_Start->BundleConfig.cs BundleConfig.RegisterBundels, add the following line:

bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(  
 "~/Scripts/jquery.unobtrusive\*",  
 "~/Scripts/jquery.validate\*"));

Part of ASP.Net MVC. It uses jQuery heavily.

* 1. Unobtrusive JavaScript: instead of injecting the JS into the HTML files, they are kept completely separated. The JS code is kept in .JS files and the HTML markup uses attributes to update the various elements that will be affected by the AJAX.
  2. The key for using AJAX is the Ajax helpers:
     1. @Ajax.ActionLink
        + Anchor tag with asynchronous behaviour
        + Parameters (overloaded):
          1. Link text
          2. Name of the action
          3. Options – rather than sending a long list of parameters, we can send an options object with any number of names parameters such as:

Error handling

How to send the request

What happens with the result

Confirmation dialog

And more

* + 1. Ajax Forms:  
       An asynchronous form
       - Initialised with Ajax.BeginForm
       - Parameters (overloaded):
         1. Action
         2. Controller
         3. Options

Insertion mode

MttpMethod (get/post)

Error handling

Target element

More

#### Raw HTML

When we need to call WebApi actions from our ajax.

## How To Build an Asp.Net Application

### New Project

1. New->Project->ASP.NET Web Application -> MVC
2. Create new model and annotate data
3. Create Controller (full MVC one)

### New Page

For every page in your application:

* + Model
    - Which model does it belong to? Do you need to create a new model? If yes, see Views
    - Set Data Attributes on the model’s properties for validation
    - Do you need to load/update this model from the database? If yes – add:  
      public DbSet<MyModel> MyModels { set; get; }  
      In the DbContext class of your application (e.g. IdentityDbContext)
  + Controller:
    - Create the controller or add a new action to an existing controller
    - Do you need access to the database in this controller? If yes – add:  
      private ApplicationDbContext \_context;  
      Remember to initialize/Dispose of it in the constructor/Dispose functions.
  + View
    - Create the new View for this page under Views/{Controller}/{Action}.cshtml
    - Does your view use just one model or does it need to process information from multiple models? If it needs information from multiple models – consider adding a new ViewModel that contain all the required data to pass to the view.

### Form Validation

In order to have form validation:

#### Client-Side Validation

Benefits:

* + 1. Immediate feedback to the client
    2. No waste of bandwidth and server-side resources

Limitation:

* + 1. Can’t prevent an attacker from building and sending a malicious http request

In order to use client side validation you should add the following @section scripts to any form that you want to validate:

@section scripts  
{  
 @Scripts.Render("~/bundles/jqueryval")  
}

Note: This validation will **only** work with the **standard Data Annotations** defined in the corresponding model. It will **not** work with any custom-validation on the **server side**.

In order to support custom validation on the client side, you will need to add it in javascript code. However, if you do – it will lower the maintainability of your code because every time you’ll need to change your code, you will have to do it in the server code and the client code!!

#### Server-Side Validation:

NOTE: the server-side validation main role is security – to prevent sending malicious data and it should always be on top of the client-side validation and not instead of it!!

1. Use data annotations on the model’s properties
2. In the Submit action: add:  
   if (!ModelState.IsValid)  
   {  
    .. re-display the same form  
   }
3. In the form view, for each field that needs to be validate, add:  
   @Html.ValidationMessageFor(m => m.Customer.Name)   
   Note: in order for the error message to be red, we must define this class type (field-validation-error) to be red in the site’s main style sheet.
4. Use @Html.ValidationSummary(<exclude property error from summary>,”error message”)  
   to display a list at the top of the page with all the validation error (or only those that aren’t already displayed in individual properties)in the form.

#### Custom Server-Side Validation

When we have more complex validation we need to implement, we’ll need to implement and custom validation. In order to do this:

1. Define a new public class under Models that derive from ‘ValidationAttribute’
2. Override its IsValid function:   
   protected override ValidationResult IsValid(object value, ValidationContext validationContext)  
   {…}
   1. If valid: return ValidationResult.Success;
   2. If invalid:   
      return new ValidationResult(“error message”);
3. In the model, on the fields that require this new type of validation, add a data annotation:  
   [CustomValidation\_MyValidation]
4. In the view, in order to see the error messages – add the @Htm.ValidationMessageFor helper function.

### Change Theme

* + Find free bootstrap themes in [www.bootswatch.com](http://www.bootswatch.com)
  + Download the new theme’s bootstrap.css file and save it (rename) under your Content folder.

### Building a Feature End-To-End

1. Understand the problem you’re trying to solve. Define the requirements
2. **Back-end first approach:**
   1. Define:
      1. Use Case: e.g. movie rental
      2. Input e.g. Customer and Movies
      3. Output e.g. none
   2. Define an action that your user is going to call and decide where you want to implement it:
      1. MVC controller – if you need to return a markup from the server
      2. API controller – if you need to return data to be processed on the client’s side.

We chose API controller because we just need a confirmation notification on the client side that will be generated in JS.

* 1. Create the action (and controller?) and the DTO for passing the data. You can throw NotImplementedException() for now.
  2. Domain Modelling – what do we need in our domain model to support this? Can use UML to understand the requirements and plan the solution.
  3. Add Model (If required):
     1. Create a new model in the models’ folder
     2. Add all required properties and data annotations  
        Note: you don’t need to add the Id and Model for every other table you need to reference. Entity is smart enough to do this for you automatically.  
        So, for foreign key, you don’t need to add:  
        public int CustomerId…  
        public Customer Customer…  
        Instead, you can simply add:  
        public Customer Customer  
        and entity will generate Customer\_Id foreign key automatically!
     3. Add the new DbSet to ApplicationDbContect
     4. Add-migration…
     5. Update-database…
  4. Change existing model (if required):
     1. Add the properties
     2. Add-migration
     3. Add (if required) Sql command for initializing the new fields to meaningful values.
     4. Update-database
  5. Implement the Action:
     1. Implement the happy path and do a sanity test using Postman
     2. Analyse and implement the edge cases.  
        Note:
        + If something should never happen (e.g. user choosing a non-existing customer from a drop-down list), we can let the application throw an exception (e.g. \_context.Customers.Single(…) and we don’t need to check and handle this error specifically.
        + If something can cause data corruption in our database (even if malicious user) – we have to apply defensive programming to avoid the problem.

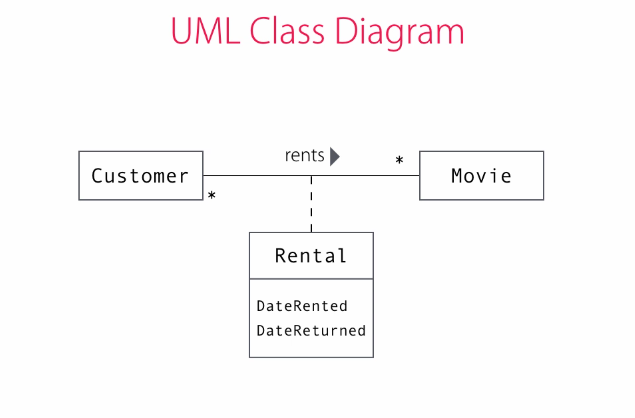
**Front End:**

* 1. Add the MVC Controller to prepare and serve the view for activating the action.
  2. Add the related MVC view with the new action
  3. Connect the new view and controller to your site (e.g. an item in the navigation bar or start from another action?)
  4. For new forms:  
     Add front-end validation:
     1. Add @Scripts.Render("~/bundles/jqueryval") to the new view
     2. Connect the form-submission to a new customer-validator:  
        let validator = $(“#<form-id>”).**validate**({  
         **submitHandler**: function() {  
         .. add the submit-form functionality post-validation.  
         }  
        })

### Example: Add Movie Rentals – User Story: Add new rental

* + Problem: the worker will identify the customer and will create a new rental form for this one customer with all the movies they rent.  
    Each movie in the rental list will need to have date rented, due date and date returned fields.

Back End 1st Approach:

* + Input: customer and a list of movies
  + Output : ok or failed
  + We’ll implement API controller since the OK screen can be rendered on the client side easily.
  + Created a new “RentalsController” under controllers\api.
  + Created a new Post action: PostNewRentals that receives a new DTO with the customer Id and a list of movies Ids.
  + Domain modelling:  
    

We need a one-to-many relationship between customer to movies and vice versa. This is an association. This association also need some additional data like date rented, due date and date returned.

Therefore, we need an Association Class to model this.

Convention: we should call out association class according to the association it represents. In this case - Rental

## ASP.Net Web API

The ASP.Net framework that allow us to build a Web API service.

ASP.Net Web API framework has the same general structure as the APS.Net MVC framework, with a few small differences.

In ASP.Net Core, MS united the MVC and Web API framework into one.

### Data Service == Web API

A web service that doesn’t generate and sends a static HTML (like an MVC app ) but instead, generate the data and send it, leaving the markup generation to the client side.

#### Benefits of Generating the Markup on the Client Side

Instead of creating the html on the server and then sending it to the client side, the server can also return just the data and leave generating the markup to the client side. The benefits of this approach are:

* + 1. Less server resources (improve scalability)
    2. Less bandwidth (slightly improve performance)
    3. Support for a broad range of clients – can send the same data to desktops, mobile, other sites etc.
    4. Note: Data Service don’t just create new data, they can also be used for modifying the data we have.

### Restful API

**Re**presentational **S**tate **T**ransfer (REST) – built on the following HTTP requests that together with the requested URL, define the requested operation. This REST protocol is stateless in that the server doesn’t need any additional information from the client other then the request type and the url. The most common request types are:

* **GET** – to get the data associated with the url. Example: /api/cutomers (for index) or /api/customers/{id} (for specific customer)
* **POST** – to create a new data. The new data required fields are sent as part of the request. For example: creating a new customer with the url: /api/cusomers
* **PUT** – update an existing data. For example: /api/customers/{id} to update customer with {id}
* **DELETE** – Delete and existing record with the url. For example: /api/customers/{id}

### Adding Web API to MVC App

1. Create a new folder: Controllers\Api
2. On the new folder -> right click -> Add -> Create a new controller -> Web API 2 Controller – Empty : {model}sController.cs   
   For example: CustomersController.cs
3. In the new controller:  
   All RESTful API methods need to return IHttpActionResult

See: <https://docs.microsoft.com/en-us/dotnet/api/system.web.http.apicontroller>

* + **GET** –   
    return Ok(IEnumerable or a single object)
  + **POST** – with [HttpPost] – get an object, validate it and add it to the database. Return the **URI** (Unified Resource Identifier) of the new object.  
    Note: if you call this method Post<object> - the framework will automatically treat it as an [HttpPost] action. However, if you change the method’s name in the future, it will break your code so it’s better to always include the [HttpPost] anyway.  
    return Created(new Uri(Request.RequestUri + "/" + customer.Id), customerDto);
  + **PUT** – with [HttpPut] can return either void (save bandwidth) or the updated object. IN both cases, return Ok(…);
  + **DELETE** – with [HttpDelete] – return Ok();

**Note**:

* + The Web API framework will translate and return the data, depending on the ‘Content-Type’ in the http request:
    - If the request did not include a ‘Content-Type’ field, it will return the data in xml by default.
    - To set the data type to Jason, include   
      Content-Type = application/json  
      in the header
  + Additional ApiController helper functions:
    - return BadRequest() – for returning error on bad request
    - return NotFound() – when record not found

### Data Transfer Objects (DTOs)

We should **never** return Domain Model Objects to the client for a number of reasons:

1. Interface stability – we don’t want to have to change the data service’s API every time we change the structure of our internal domain objects! The Web API is our public contract and as such it should change at a relatively slow pace. Much slower than our internal changes.
2. Security Hole – a hacker can include hidden fields of out domain object that shouldn’t be updated (e.g. Id in customer). This can cause our application to accidently change unexpected fields.   
   DTOs, we don’t have fields for properties we don’t want to set.

For these reasons, we create and use DTOs for transferring data from the server to the client and back.

**Note:** Changing DTOs is costly and can break our service therefore, it should be done carefully and in a planned way, using API versioning to avoid breaking existing clients’ code!

#### Adding DTOs

1. Under the main application folder -> Add new folder Dtos
2. In the Dtos folder add new class <model object>Dto and include all the field we need in this DTO.  
   Note: **never include** any **domain model** object **in the DTO** class. If it has a member that is a model object, either create another DTO for it, or remove it from the main DTO completely (if it’s not needed). This way, the DTOs are completely de-coupled from the domain model.

### Camel Casing

Since our C# properties have PascalCasing and JavaScript which consumes the data that we return from our REST API works with camelCasing, we need to translate the casing when we receive/return JSON objects through our Web API.

In order to do this:

In WebApiConfig::Register, add the following lines:

using Newtonsoft.Json;  
using Newtonsoft.Json.Serialization;

…  
var settings =   
 config.Formatters.JsonFormatter.SerializerSettings;

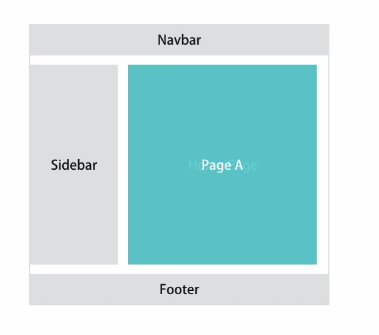
settings.ContractResolver =   
 new CamelCasePropertyNamesContractResolver();

settings.Formatting = Formatting.Indented;

### Testing Web API

Install Postman – REST Client Extension to your Chrome

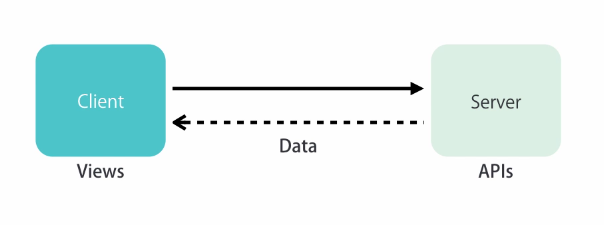
## Single Page Application



An application that has one general markup for the general page data with the content section being the only one that changes between pages.

The content is retrieved from the server using Web API calls which return the data to be rendered on the client side.

This type of application usually doesn’t require the Razor engine since the server is not the one rendering the markup.



Server Side:  
\* ASP.Net Web API

Client Side:  
\* Angular  
\* Backbone  
\* Ember  
\* React.js

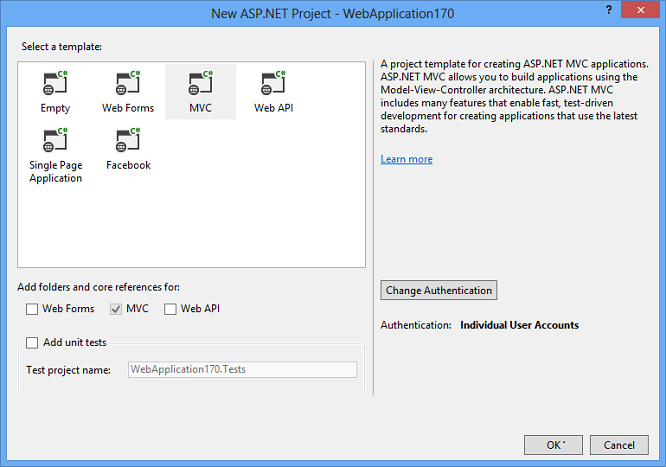
### Advantages:

The user experience is smoother since the application doesn’t need to get full pages every time from the server and it only changes the part of the page that requires changing.

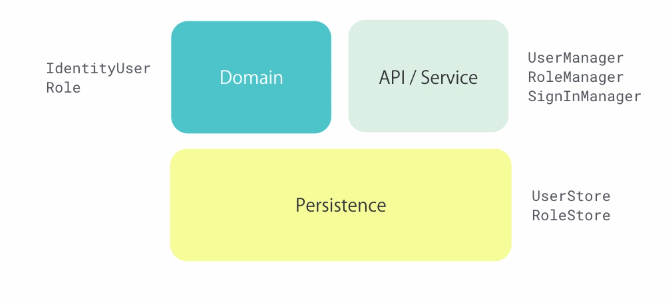
AJAX is one of the technologies heavily used by Single Page Applications (SPAs). As the user clicks on a link, we call the server in the background, get the data for the target page, generate the mark up on the client, and use JavaScript to refresh the content area.

## Identity Framework

Works with ASP.Net to handle user’s management and logins.

To create an MVC project with Identity, choose ‘Individual User Accounts’ in the new ASP.Net Project dialog box:  


### Architecture



e.g. database, files etc.

### Usage

#### [Authorize] and [AllowAnonumous]

[Authorize] attribute- can be added to actions, controllers or globally:

1. On actions – will call the action only for logged in users. If the user is not logged it, the framework will redirect the user to the log-in page.
2. On controllers – will limit all actions of this controller to logged users:  
   [Authorize]  
   public class MoviesController : Controller {..}
3. Globally – if most of our application needs to be available to looged users only. In order to enable this:
   * In App\_Start\FilterConfig.RegisterGlobalFilters, add:  
     filters.add(new AuthorizeAttribute());
   * To allow access to specific pages/controllers for non-logged in users, add the attribute:  
     [AllowAnonymous] to them.
4. Each action in our application has to include   
   **[Authorize(Roles = <list of roles>)]** to make sure that we don’t have security holes.

#### Users and Roles

<https://gooroo.io/GoorooTHINK/Article/17333/Custom-user-roles-and-rolebased-authorization-in-ASPNET-core/28352#.XJq4aqIRXct>

When creating a store, we have to have at least one user with manager permissions in our store when we 1st deploy our application so that they can start adding products to the store.

So we need to create the role and seed one user in advance.

Convention: Roles should be named after what they can do and not using generic labels like Manager or Admin. This makes it easier in larger and complex application to keep track on what each page/action permissions are.

#### Seed Users and Roles – Through Code

<https://gooroo.io/GoorooTHINK/Article/17333/Custom-user-roles-and-rolebased-authorization-in-ASPNET-core/28352#.XJq4aqIRXct>

1. This (and other) examples didn’t work for me – see my code in AspDotNetCourseApp. I think it has to do with the password hashing. But I don’t have time to dig more into it. So instead, I run this code and then manually add a new user and connect them to the admin role.

Note: Defining the default admin that can manage the reset of the users is something that is done one time only per project. So it’s probably not worth so much trouble and I can just create it manually as bellow or better yet:

1. Create the roles automatically like in this section
2. Add the users through the running program like below
3. Connect the users to the roles like below
4. Create the new migration and run it like below.

#### Seed Users and Roles –Manually (Create Migration)

1. Run the application and create the new users through the ‘register’ dialogue.
2. Open the database explorer:
   1. ->AspNetRoles and add the required roles straight into the database.
   2. ->AspNetUserRoles – connect the users to their appropriate roles
3. Add a new migration:  
   PM> add-migration SeedUsers
4. In the migration Up() function, add:  
   Sql(@”…”);

Where … contain all the Sql script command you take from the database explorer: for each of the tables:

* + AspNetRoles
  + AspNetUsers
  + AspNetUserRoles

Mark the lines with the data -> right click -> Script. Copy the SQL instructions and paste it in the Sql(@”…”); command.

1. Delete all the data (see previous point) from the database
2. Run:

PM> update-database

To run the migration.

#### Check Roles

1. User.IsInRole(..)

If(User.IsInRole(“CanManageMovies”))  
{  
 // render the edit version of the view:  
 return View(“List”);  
}  
else  
{  
 return View(“ReadOnlyList”);  
}

1. User.Identity.IsAuthenticated  
   {…}

### Add Profile Data

1. Add the new properties in the Identity Model:  
   Models\IdentityModels.cs : ApplicationUser
2. Add the property to RegisterViewModel and to ExternalLoginConfirmationViewModel (if enabled)
3. Add the new property to the view: Views/Account/Register.cshtml  
   and to ExternalLoginConfirmation.cshtml (if enabled)
4. In the Post action of the Register form in AccountController.Register and in AccountControoler.ExternalLoginConfirmation – add the new property to the new ApplicationUser.
5. Add a new migration for the change in model (add-migration…) and update the DB (update-database)

### Social Logins – OAuth

Most of the big social networks (e.g. Facebook, google etc) uses the OAuth authorisation protocol.

#### Open Authorisation (OAuth)

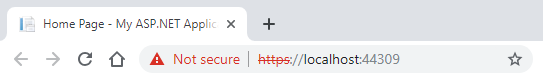
When a user asks to login to our site with a social media account, our site will:

* + - 1. Open a secure channel (HTTPS using SSL) with the social media platform (e.g. Facebook)
      2. Authenticate our app to the social media platform
      3. Get the authorization of the user to login using their social media account
      4. Get the details (we have access to) on the user from the platform.

#### Using Social Logins in Our App

##### 1.Enable SSL (for HTTPS with social media site)

1. Click on your project in solution explorer -> press F4
2. Set ‘SSL Enabled’ to True
3. Copy the address in **SSL URL** – this is the address we use to establish a secure connection with our site.
4. Open the project properties -> Web -> Project Url and replace it with the SSL URL you copied.

Now when we’ll run our app, it will work through the https port. We will receive errors/notifications since our certificate is invalid:  


To deploy our app in production, we will need to get a proper certificate from our site hosting provider.

1. In RegisterGlobalFilters, disable support for non-secure http connections:  
   filters.Add(new RequireHttpsAttribute());

##### 2.Register our app with social media site:

1. Go to developers.facebook.com
2. For 1st time only – register yourself as a developer (free)
3. Add new app and fill in the detail
4. Go to developer Dashboard and to your app’s page
5. Copy the ‘**App ID**’ and the ‘**App Secret**’ and paste them in the boilerplate code for app.UseFacebookAuthentication in Startup.ConfigureAuth
6. Now our users can register using their Facebook account. The same can be done for google and other social media accounts.

##### 3.Update Profile Data

If we added required fields to our standard registration form, we’ll need to add it to the external registration forms. Otherwise, we will receive an exception. In order to do this:

1. In ExternalLoginConfirmation.cshtml – add the new fields (like we did in the registration.cshtml)
2. In ExternalLoginConfirmationViewModel – add the new fields (like we did in RegistrationViewModel.
3. In AccountController.ExternalLoginConfirmation – add the new fields to the new ApplicationUser constructor.

## Performance Optimizations

##### “Premature optimization is the root of all evils” – Donald Knuth.

!! Profile first. Then, if you need to, Optimize !!

### Profiling Tools

#### Glimpe

A profiling tool that uses cockies on your requests in order to profile how long they take.

* Install:  
  PM> install-package glimpse.mvc  
  PM> install-package glimpse.ef6
* Navigate to your <site url>/glimpse.axd
* You can turn glimpse on/off. Note: this works only for localhost. To profile a deployed application – you need to do additional steps.

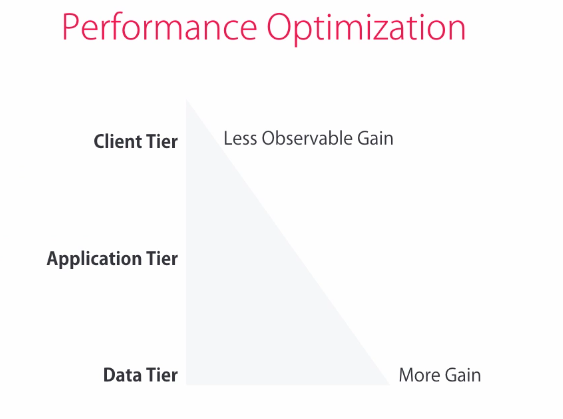
### Three Tier Architecture

Most Web applications run in a three-tier architecture:

* Client Tier – the client computer, running the front end on the browser.
* Application Tier – the server hosting our application
* Data Tier – the database Server (e.g. SQL)

### Performance Optimizations

Usually, most of the performance bottle necks exist in the Data Tier. Therefore, optimizing it usually gives us the best visible results:



#### Data Tier Optimizations

Data Tier optimization can be split into two:

##### Schema Issues:

* Make sure that your tables have:
  + Primary keys (see .PrimaryKey in the migration files)
  + Relationships
  + Indexes on columns that we use to filter results in our queries. (see .CreateIndex in the migration files)
* Schemas with Entity-Attribute-Value (EAV) pattern can cause a lot of performance issues. In this pattern, instead of having concrete tables like movies, customer etc – the schema has only a few tables called Entity, Attributes and Values and some other supporting tables.

The reason for using this pattern is to avoid schema changes.

However, the cost of using this pattern is huge:

* + Can’t use Entity Framework or any other ORM.
  + Have to write all the queries by hand and they usually have to be very complex, long and
  + Extremely slow

##### Queries Issues:

* Usually, Entity framework’s queries are fine. However, in more complex queries, sometimes we need to look closer at the query that the Entity Framework generates and maybe write the store procedure ourselves to optimize the query.
* Use Execution Plan is SQL Server to see how SQL server execute your query. This can show you what part of your query has the largest cost and then you can optimize it.
* Caching
* Option: create a separate, optimized database for reading data (look up CQRS). This might be good for application that usually read the data and more rarely write it.   
  Note: this approach comes with the cost of maintaining two databases in synch
* **N + 1 Issue**:  
  this can happen **sometimes** when we use Lazy loading of data from the database (which is the default Entity model, unless we use Include on our queries). Lazy loading means that the framework will only get the data of the current record and will not load the data from any additional records that our class reference. The additional records will be fetched from the data base only when we loop over the list of record and query them. So, in lazy loading, if we have N customers and we need to get the MembershipType (a reference to another record) for each of them, with lazy loading we will have up to N+1 separate queries to the database: one to get the list of customers and N to get the different possible MembershipTypes to match to the customer.MembershipTypeId for each customer.

For this reason, it is usually a good idea to **avoid lazy loading** on your application!

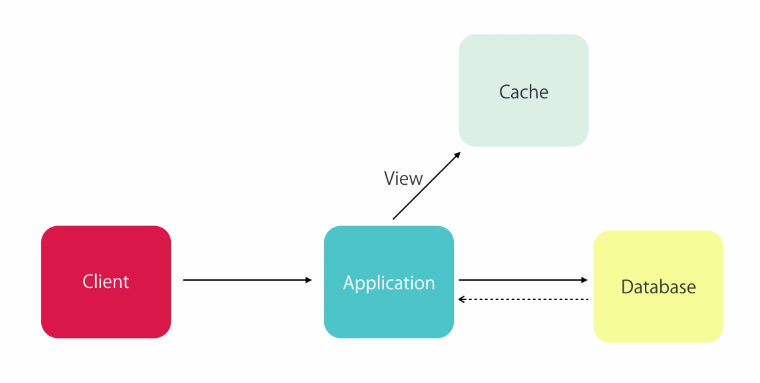
#### Application Tier Optimization

##### Output Caching i.e. rendered HTML

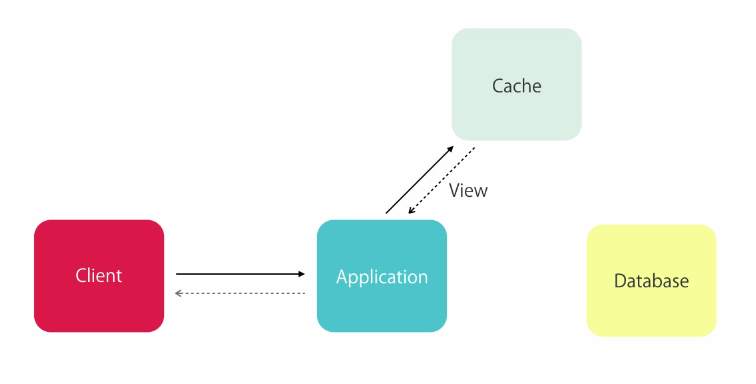
This usually gives the best performance gain in this tier.

Output caching means that the MVC framework will save the rendered HTML in the cache after the first time it creates it and will serve it from the cache instead of re-calculating it when we visit the same page again (as long as the cache wasn’t cleaned and the data didn’t change).

1st call:



Next calls:



To enable Output Caching on an action:

* add the   
  **[OutputCache(** Duration = <time in seconds>,  
   Location = OutputCacheLocation.Server/Client,  
   VaryByParam = <specific param e.g. “genre” or “\*” for all> **)]**  
  decoration to the action or the controller.
  + VaryByParam – will hold a different cache for each param
  + Location – if the view is specific to one client – we’ll put the cache on the client. Otherwise, we’ll put it on the server.

Notes:

* The downside of caching is serving stale data. Therefore, don’t optimize unless you need to and when you do, consider the parameters you use carefully!
* Sometimes the users’ browsers can cache the data and cause a bad user experience where the page doesn’t refresh. In order to prevent this, you can actively disable the caching on specific actions/controllers by adding:  
  [OutputCache(Duration=0, VaryByName=”\*”, NoStore = true)]

##### Data Caching

If, after profiling your app, you need to cache some data:

using System.Runtime.Caching;  
…  
if (MemoryCache.Default["Customers"] == null) {  
 MemoryCache.Default["Customers"] =  
 \_context.Customers.Include(  
 c => c.MembershipType).ToList();  
}

var customers = MemoryCache.Default["Customers"] as IEnumerable<Customer>;

return view(customers);

If VS doesn’t recognize system.runtime.caching , this means that you need to add a reference to this assembly:

1. In solution explorer, right-click on "References"
2. Select "Add reference"
3. From left side menu select "Assemblies"
4. Look for (or filter) and add System.Runtime.Caching.dll

Notes:

Using Data Caching comes with its own complications and pitfall. Therefore:

* Use it only when needed and after profiling
* Use it for displaying data and not for changing data only.

##### **Release Build – Do Always (no profiling needed)**

Deploy with release build to improve performance

##### **Disable the Session Data – Do Always (no profiling needed)**

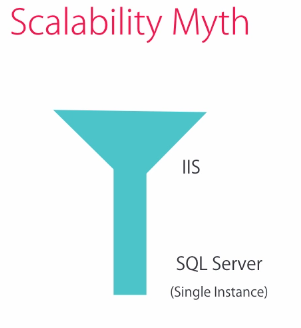
Make your web apps stateless to improve scalability. In order to disable the session state on your app:

1. Open Web.config
2. In <sestem.web> add:  
   <sessionState mode=”Off”></sessionState>

##### Asynchronous Operations

Note: Using async and await on operations that take time, will allow us to scale out application better because a thread will not need to wait for the DB to respond before accepting new requests from the router. However:

* They will no improve performance because every request from the server will still have to wait until we get the data form the database/network before returning to the user.
* The scalability gain will also be minimal because our single DB server is out application real bottle neck:



* To get real scalability, we’ll need to use SQL cluster, NoSql DB or SQL Azure.

#### Client Tier Optimizations

Client Tier optimizations are based on two main principles:

* Reducing the number of requests
* Reducing the amount of data sent in each request

These principles can be applied to:

* DTO – avoid returning unnecessary properties
* JS and CSS–
  + Put all scripts near the end of the html so that the page will load as soon as possible and not hang waiting for the script
  + Combine and compress the required JS and CSS files using ASP.Net bundles. This happens automatically by ASP.Net on our release builds, using the bundles we define in BundleConfig.cs.
* CSS
* Image

## Debugging

### Printouts

* Trace.WriteLine – will write to the debug output when running the application both in debug and release modes.

### Logging Unhandled Exceptions – Using Elmah

PM> install-package Elmah

Elmah plugs into the request processing pipeline so whenever it sees an unhandled exception coming back from any request, it grabs its details and stores them.

By default it stores the unhandled exceptions in the memory. To see the exceptions:  
locally, you can navigate to <application url>\elmah.axd

In order to be able to access this data remotely, you need to add the following configuration:  
In Web.config, under:   
<location path=”elmah.axd”…>  
 <system.web>  
 <httpHandlers>  
 …  
 </httpHandlers>  
 add:  
 <authorization>  
 <allow roles=”<roles that can access the data>” />  
 <deny users=”\*” />  
 </authorization>

* Roles – the roles that can access this data, as defined in the AspNetRoles table. For example: roles=”Admin”
* Instead of roles, you can also user users=”least of user names” for example:  
  <allow users=”admin.sarah@gmail.com,user1.sarah@gmail.com” />

We can also configure Elmah to store the exceptions in SQL server so they won’t be cleared every time the application starts. See Elmah documentation on how to do this.

## Security

### Custom Error Pages

When our application throws an exception, the default error page displays a lot of internal information we don’t want users (or hackers) to see.

#### Enable Custom Error Page:

**Application Unhandled Exceptions:**

In Web.config, under <system.web> add:

<**customErrors mode=**”RemoteOnly”> </customErrors>

* Mode=”RemoteOnly” – the custom errors will be displayed only when browsing the app remotely (e.g. deployed) but when developing (i.e. browsing through localhost), it will still display the full debug output.
* Mode=”On” – will always display the custom errors.
* Customize the View:

The custom error view that will be displayed is: Views/Shared/Error.cshtml

So, you can customize it as well.

* How it Works:

The HandleErrorAttribute handler, which we registered in ‘RegisterGlobalFilters’, runs after every action and if the action had an unhandled exception, it will catch it and rendered the custom Error.cshtml view

**Application returned error (no exception):**

Add <error> tags inside the <customErrors> tags above:

<customErrors mode=”RemoteOnly”>   
 <**error statusCode=**”<status code>”  
 **redirect**=”~/<custom>.html” />  
</customErrors>

* <status code> - can be any status code e.g. 404 (HttpNotFound())
* The custom.html file – a custom html file that will be served any time our application returns this status code.
* Note: for page not found error (404) – if the IIS is the one returning the 404 error, without sending the request to the application (e.g. when a user tries to access an image file), it will return its standard detailed 404 error page.  
  if we want to customize IIS error pages, we’ll need to add another configuration to the server:

**IIS Returns an Error (request didn’t reach our application):**

* under <system.webServer>:

<httpErrors errorMode=” DetailedLocalOnly”>  
 <remove statusCode=”<error code >” />  
 <error statusCode=”<error code>”   
 path=”<custom>.html”   
 responseMode=”File”/>  
</httpErrors>

* + errorMode=”Custom” will display the custom error page for all users (both local and remote)
  + errorMode=”DetailedLocalOnly” – will display the detailed message to local users (e.g. development) and the custom page to remote only.
  + Remove – tells the IIS which error code to remove from its default handling
  + Error…path is the static error page we created
  + ResponseMode - we should set this to “File” otherwise it will change the returned error code which effect search engines performance badly.

Example:

### Cross-site Request Forgery (CSRF)

When a malicious web-site uses a valid session that a legitimate user has with a server to send requests, using the user’s session.

To protect against this attack:

* + In any form – add   
    @Html.AntiForgeryToken()  
    at the end of the form (just before the submit button).

This will create a token and put it on the form as a hidden field and it will also save it on the user’s computer as a cookie.

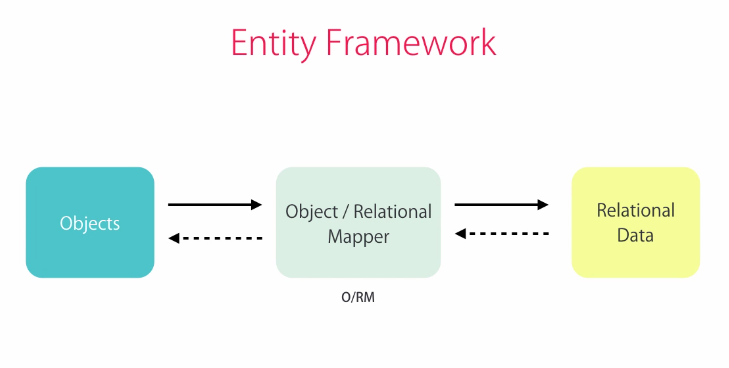
* + Add the annotation: [ValidateAntiForgeryToken] to the form’s submit action in the corresponding controller.

This method also prevents C-Serf attacks.

Since the token exist in a hidden field only, the attacker will not be able to see it. Therefore, they will not be able to send it with their forms and whenever they send the form, the server will compare the received token with the expected one in the field \_requestVerificationToken. If they don’t match, it will throw a HttpAntiForgeryException and will not process the request.

## Entity Framework

ORM (Object-Relational Mapper) to work with a database using .Net objects.



* + DbContext – define the <<database>> we’re using – usually one per app.  
    It has 1 or more
    1. DbSet – define a <<table>> in the database
       - We use LINQ to query the DbSet.  
         The Entity framework translate our queries to SQL queries to the database in runtime.
       - When we add/modify/delete records in our DbSet, Entity remembers them and when we send it the Persist command, it sends them to the SQL database.
  + There are two main workflows to work with Entity Framework: Database First and Code First

### Install Entity Framework

1. In VS, Tools-> Nuget Package Manager -> Packets Manager Console
2. PM> install-package EntityFramework
3. Restart the solution for the installation to take effect.

### DbFirst Workflow

1. Design the tables in the database
2. Let Entity Framework create the appropriate classes from these tables.

### CodeFirst Workflow

1. Write the code (Domain classes)
2. Have Entity Framework create the appropriate tables from them.

CodeFirst Workflow is **the recommended approach** because:

* + 1. Increased productivity – creating database tables with a db designer takes much longer than just writing the DB classes in code.
    2. Full versioning of DB – our code is saved in Git and is therefore, very easy to migrate to new versions of the DB.
    3. Much easier to build an integration test db.

Untrue myths about CodeFirst Workflow:

* You can use it on greenfield projects only. This is not true. You can add any new tables in the CodeFirst workflow and enjoy its benefits for them.
* It doesn’t give us full control over db. This is not true – search for this.

#### Enable Migrations

Required for every new project that uses migrations:

1. PM> enable-migrations
2. VS will create a new folder ‘Migrations’ in your solution root folder.

#### Create Migration

PM> add-migration InitialModel

Note: you must have a DbContext class (or derivative) in your solution, otherwise, you will receive a “context not found” error.

If you need to overwrite the original migration:

PM> add-migration InitialModel -force

Creating a new migration in this way, create a new .cs file under Migration folder in your solution root folder. This cs file (and its folder) are named according to the exact date and time when you created this migration. This allows you to save this migration in your source control and to run all migrations in order from the creation of the database until the very last migration (or any other sub-set in between). It also allows you to see either all databases or just testing/production databases in code and to save this as part of your source control as well!

#### Add Models

In order for the Entity Framework to know and migrate our classes, we need to add their respective DbSet as public properties in our main DbContext class:  
public DbSet<name of our model class> <name for table> { get; set; }

For example:

public DbSet<Customer> Customers { set; get; }

#### Run Migration and Update Database

PM> update-database

This will create the database file (.mdf) under App\_Data

To see the SQL commands – user the flag -Verbose

#### Create Small Migrations!

In order to avoid problems, change the database/models in small steps (like you would with commits). For every small change in the modems, create a new migration with a meaningful name and run it to make sure that there are no errors:  
PM> add-migration AddIsSubscrivedToCustomer  
PM> update-database

#### Seeding the Database

Since we are working on code-first model, we should never open the database and change the DB directly. Instead, we should create a new migration:

PM> add-migration PopulateMembershipTypes

And then, change the code in this migration class.Up() to add the required records. For example:  
in Migrations.PopulateMembershipTypes.Up()  
{…  
 sql(“INSERT INTO MembershipTypes (Id, SignUpFee, DurationInMonths, DiscountRate) VALUES(1,0,0,0)”);  
…  
}

* Note: For playing around while we develop, we can add some test data directly into the tables through the SQL Server Explorer. We just don’t want to do this for anything that should be repeatable – either test suites or production data.

**Seeding on Startup**

To seed the database with initial required data, we can test for it and seed it when the application start:

In Application\_Start():

…

// Seed User Roles if undefined:

var identitySeed = new IdentitySeed();

identitySeed.CreateRolesAsync(new ApplicationDbContext()).Wait();

In IdentidySeed:

public async Task CreateRolesAsync(ApplicationDbContext context)

{

var userManager = new UserManager<ApplicationUser>(

new UserStore<ApplicationUser>(context));

var roleManager = new RoleManager<IdentityRole>(

new RoleStore<IdentityRole>(context));

IdentityResult roleResult;

foreach (var roleName in \_roleNames)

{

var roleExist =

await roleManager.RoleExistsAsync(roleName);

if (!roleExist)

{

roleResult = await roleManager.CreateAsync(

new IdentityRole(roleName));

}

}

}

Note: for adding users - because of the password hash, we can’t add them directly into the table in this way. Therefore, we should:

* + 1. Register them as new users through the normal app interface (Home Page -> Register)
    2. Add them to the AspNetUserRoles table with their appropriate roles (e.g. Admin)
    3. Create a new migration for the seeding process: SeedingDefaultUsers
    4. In the SQL Server Object Explorer -> Right Click all rows we need -> Script
    5. Copy all the SQL statements and paste them into the migration’s Up() function as:

Sql(@"

…

INSERT INTO [dbo].[AspNetUsers] …

…

INSERT INTO [dbo].[AspNetUserRoles] …

");

#### Overriding Database Conventions

To override the database default data types, we can use Data Annotations when defining the Models. See Data Annotations

#### Clear Database

If you want to start your database from scratch run the following commands:

PM> Update-database -TargetMigration:0

// Normally your DB is empty now since the down methods were executed.

PM> Update-database

// This will recreate your DB to your current migration

### Database Structure

* Double click the file (.mdf) will open it for database view.
* Under Tables\\_MigrationHistory – this table is managed by VS. We should not touch it. It saves the history of all the migrations that the DB went through.

### Query the Database

In order to query the database in our code we need to connect our class to the ApplicationDbContext. For every class that needs access to the database:

1. Add the following private member:

private ApplicationDbContext \_context;

1. Add a constructor and initialize the internal member from it:

public CustomersController()  
{  
 \_context = new ApplicationDbContext();  
}

1. Make sure to dispose \_context in the Dispose function:

protected override void Dispose(bool disposing)  
{  
 \_context.Dispose();  
 base.Dispose(disposing);  
}

#### Deferred Execution

When we assign the \_context to a model, Entity framework will not query the database! It will only query the database when we start to iterate over the \_context object.

For example:

// Entity framework will defer querying the DB on this line:  
Var customers = \_context.Customers;

// It will query the DB only in the View when we iterate over customers and display their details:  
return View(customers);

#### Lazy Loading and Eager Loading

By default, the Entity framework will only load the specific record type we asked for, without loading the data for any reference included in the record type. For example:

var customer = \_context.Customers.SingleOrDefault(c => c.Id == Id);

// the next line is true. The foreign key is loaded correctly:  
customer.MembershipTypeId == 1;

// the next line is null – the reference type for the 2nd  
// model is not loaded by default!  
customer.MembershipType == null;

This is called **Lazy Loading** To work around this, we use **Eager Loading** to include the reference data we want Entity to include using the Include() call:

Using System.Data.Entity;  
…  
var customer = \_context.Customers.**Include**(c => c.MembershipType)….

now:  
customer.MembershipType contains the record from the MembershipTypes table and not null.

#### Persist Changes

When we change the data in our ORM, the Entity framework will mark this change but will not save it into the DB! In order to save all these changed to the DB, we need to call:  
\_context.SaveChanges();

All the changes that the framework recorded at the time of the .SaveChanges call will be translated to SQL statements at run time and then run them all on the DB.

Note: All the SQL statements are wrapped in one DB transaction. Meaning that **either all** changes will be **saved** to the DB **or none** of them will. If any change to the DB fail, it will revert all the other changes and the DB will remain consistent with the way it was before the .SaveChanges() command.

### Database Actions

* Add – to add a new record to a table:  
  \_context.Customers.Add(customer);
* TryUpdateModel(<model instance to update>,  
   <prefix>,  
   <include properties>  
  Will try to update the <model instance to update> with the @model supplied values (through the model binding).

Note: some issues with using this method for updating the DB:

* + If <include properties> is not included – a serious security hole – an attacker can add invalid properties in the query to hack the site.
  + If you rename any of the <include properties> in the future, they will not change here and the app will crash in runtime!
  + More secure alternative for using TrayUpdateModel are:
    - Manually set all parameters in the model – not very readable/friendly
    - Use a DTO as parameter and AutoMapper library to set all fields automatically.

## Deployment

### In Visual Studio

#### Publish your Source Code

Right click your solution -> Publish

* The most commonly used options are File system or FTP under IIS,FTP, etc.

Additional publish methods:

* + Web Deploy – for publishing directly to IIS site. Requires a target IIS.
* For production – use release mode. For testing or other custom – profiles (see previous section) – use the required profile.
* Note: for file release, you can’t publish the database. So you’ll need to create the SQL script for running all your migration.

#### Database Deployment

Unless you’re publishing to Web Deploy and running your code-first migrations as part of the deployment, you will need to publish your database manually:

**First Time Deployment:**

PM> update-database -script

This will create a SQL script with all our migration scripts

**Deployment Updating:**

PM> update-database -script -SourceMigration:<the 1st migration class that we want to migrate>

For example:  
PM> update-database -script -SourceMigration:InitialModel

This will create the SQL script that include all migrations starting from the class we included in the update-database command and until the latest migration in our code.

**Note:** in order to check what was the last migration run on any database, you can look at its \_MigrationHistory table.

The resulting SQL script, you will need either to give to an administrator and handles the database or to a database migration tool that works with SQL scripts.

## Tools

Web Platform Installer – allow installing Web-development related tools easily on the PC.

### AutoMapper

Allow copying fields automatically between two objects of different types.

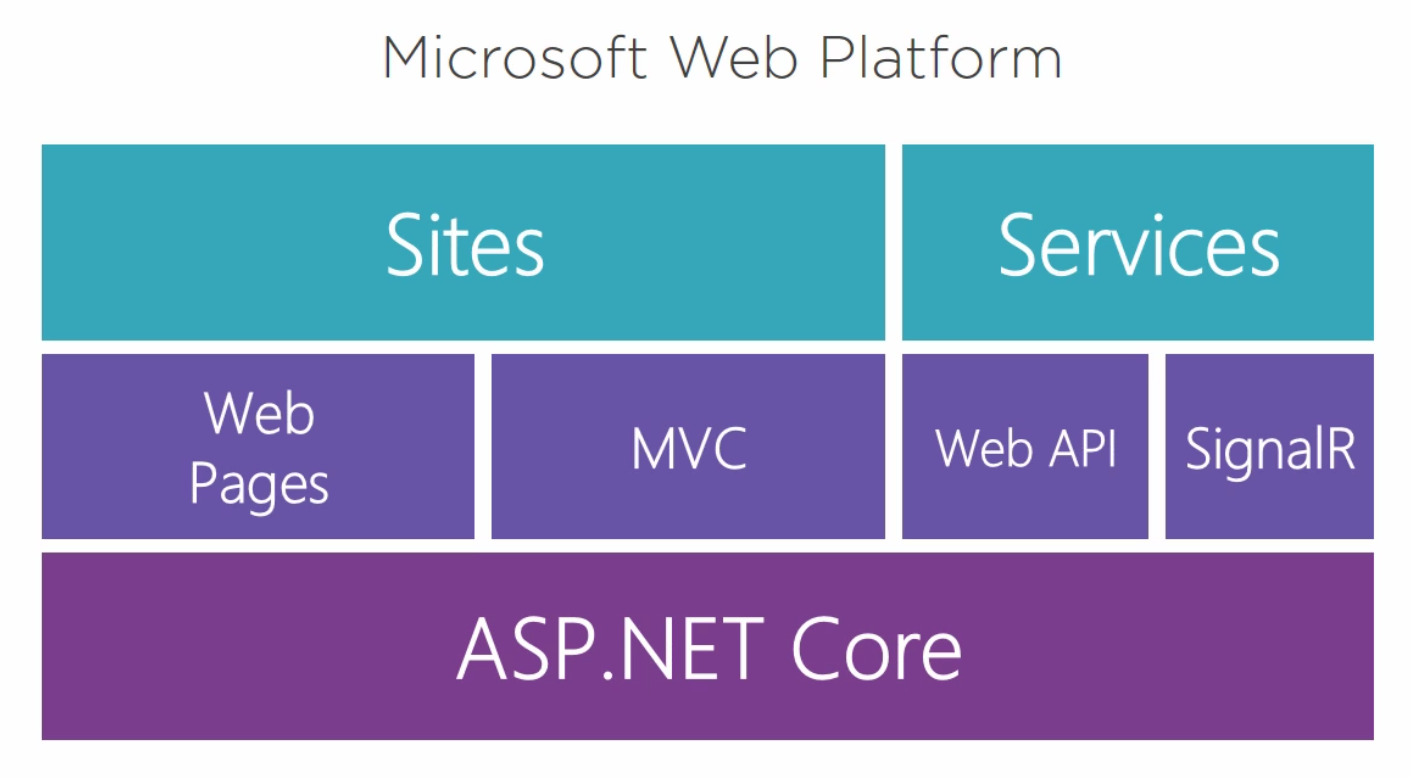
1. To install:  
   PM> install-package automapper
2. Set up:
   1. In App\_Start directory -> create new class MappingProfile : Profile
   2. In its constructor, add:  
      CreateMap<fromObjectType,toObjectType>();  
      you can also add .ReverseMap(); at the end to support mapping both ways.  
      for every object you want AutoMapper to map.   
      When the constructor runs, AutoMapper will go over all the mappings defined here and will map them according to their properties names.
   3. In Application\_Start, add:  
      Mapper.Initialize(cfg => cfg.AddProfile<MappingProfile>());  
      This will call the MappingProfile constructor on our application startup.
3. Usage:
   1. As input for Select (or other functions requiring a selector):  
      Mapper.Map<toObjectType>
   2. To directly map between the objects:  
      var customerDto =  
      Mapper.Map<toObjectType>(customer);

Or:

Mapper.Map(fromObject, toObject); to update the ‘toObject’ with the data in the ‘fromObject’

## ASP.Net Core

New->Project->ASP.NET Web Application ->ASP.Net Core->Web Application



ASP.Core was re-developed from scratch and allow the same as ASP.Net but multiplatform. Web Forms are not supported in ASP.Net Core

ASP.NET Core (ASP.NET 5) is a lean framework for building web and cloud applications. ASP.NET Core is fully open source and available on GitHub

### Install APS.Net Core

1. Browse to dot.net
2. Download .Net core SDK and install it.
3. To verify that its installed:  
   in command line: > dotnet –version

### .Net Core Command Line

dotnet -–help – to get help on commands.

> dotnet new console – create a new console application in the current directory. It also runs the

> dotnet build – build the project

> dotnet run <optional arguments> – will run what is in the current directory

> dotnet add package <package name> – add a Nuget Packet dependency to our project.

### Start a new .Net Core Project

1. File -> New -> Project -> Web or .Net Core -> APS.Net Core Web Application.
2. Choose the version of .Net Core
3. Choose MVC for starting point.
4. Choose User authentication option

### .Net Core Project from Scratch

1. Create a new .Net Core project and choose ‘empty’ instead of MVC.

### .Net Core Project – Structure

#### wwwroot

A directory that holds all the files that you want to serve to the users. It is treated as the root of the web server and allow direct access to files there.

#### Startup.Configure

This method is called by the runtime to configure the HTTP request **pipeline**.

It will determine what middlewares will be called for every new web request received by the app.

NOTE: The ORDER in this function MATTERS! This is exactly the order in which the different middlewares will be called whenever the app receive a new request!!

#### Controllers, Models and Views

Will use the same files structure as traditional ASP.Net projects:

* + - Controllers live in folder Controllers:  
      Controllers/ProductController.cs
    - Views live in folder Views/<Controller>/ and named according to the controller’s actions:
      * Views for ProductController will live in Views/Product
      * Each view named for an Action in the controller

Views/Product/Edit.cshtml

* + - \_ViewImports.cshtml – contains global imports for all other views.  
      This is where we’ll put all general @using and @addTagHelper lines.

#### Views

**Forms:**

* + - Use regular HTML syntax for forms:  
      <form method=”post”>…</form>
      * the form method must match the [HttpPost] tag in the controller’s action
      * The controller will be the <ControllerName>Controller from: Views/{ControllerName}/view.cshtml

### Configuration File

The configuration file in ASP.Net Core is different than the ASP.Net xml-based configuration file. This is because the xml-based configuration file was too cumbersome and hard to extend so it was used in a very limited and simplistic way.

In ASP.Net Core, the configuration file was rebuilt to be much more flexible and extendable.

If working on a green-field project, we can use just json for all our configuration.

However, most legacy projects will have their configurations in xml, ini files, system variables and even databases.

That’s why the new IConfigurationBuilder allow adding all these types (and more) of configuration files.

We can use all of these together:

private static void SetupConfiguration(WebHostBuilderContext context, IConfigurationBuilder builder)  
{  
 // Remove the default configuration options  
 builder.Sources.Clear();

builder.AddJsonFile("config.json", false, true)  
 .AddXmlFile("config.xml", true)  
 .AddEnvironmentVariables();

}

The ConfigurationBuilder will **combine all these configuration sources** into one configuration store that we can use in a **hierarchical** way. So that if the same configuration exists in different sources, **the last source wins**.

#### Create a New Configuration File

In order to take values from configuration files instead of having them hard-coded in the code, we need to connect the configuration ability to our project:

1. Add a constructor to Startup:  
   private readonly IConfiguration \_config;  
   public Startup(IConfiguration config)  
   {  
    \_config = config;  
   }
2. The default configuration files is appsetttings.json at the root of the project.
3. To read values from the configuration file:

\_config.GetConnectionString(“<name of string>”);

#### Configuration File Commands

* GetConnectionString(“<name of string>”) – will read a string from "ConnectionStrings" object.

### Entity Framework Core

* A complete re-write of the Entity Framework
* Removed the requirement for Relational DBs. It can also work with additional types of DB like document store, graphical DBs etc.

#### Create and connect a DbContext

1. In Data folder, create a new class to represent the database:  
   class ApplicationContext : DbContext{}
2. Connect the required database type: In Startup.ConfigureServices, add the DbContext:  
   services.AddDbContext<ApplicationContext>( cfg =>  
   {  
    // See Microsoft.EntityFramworkCore for all   
    // the database types options  
    // This one is for the most common one - SqlServer:  
    cfg.UseSqlServer(“<connection string from config file>”);  
     
   });

#### Entities

* These are the mapping between the database and our code.
* They live under Data/Entities

#### Entities Properties

Entity Framework Core support two methods for setting the properties of the entity:

* Data Annotations: like in EF in ASP.Net. For example:  
  [Required]  
  [MaxLength(500)]  
  public string Url { get; set; }
* Fluent API:  
  override the OnModelCreating in your application’s ApplicationDbContext class and add it to the modelBuilder:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.Url)

.IsRequired()

.HasMaxLength(500);

}

#### Common Properties

* **HasColumnType** – to specify column types that don’t map automatically (e.g. decimal).   
  It takes the name of the type and a HasPrecision( byte precision, byte scale )
  + where precision is the total number of digits the db will store, regardless of where the decimal point falls and
  + scale is the number of decimal places it will store.

Examples:

* + Data annotation: [Column(TypeName = "decimal(5,2)")]
  + Fluent API: HasColumnType("decimal(5, 2)");

#### Connect Entities

* Every entity that we need to query directly (and not through another entity) from the DB, we need to connect to our DbContext class (database) by adding their DbSets (table) as properties to the DbContext class:  
  public DbSet<name of our model class> <name for table> { get; set; }

#### Seed Database – by Overriding OnModelCreating

For simple tables, we can use the following method for seeding the data. In the application’s main ApplicationDbContext class, overrides the OnModelCreating function:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Magazine>()

           .HasData(new Magazine{

MagazineId=2,

Name="New Yorker"

},

                    new Magazine{

MagazineId=3,

Name="Scientific American"

}

           );

}

Note:

* The OnModelCreating will add this data on the next time that we’ll create a new migration through ‘ef migrations add’ command.
* This feature is most useful for **static data that's not expected to change** outside of migrations and **does not depend on anything else** in the database, for example ZIP codes.
* If your scenario includes any of the following it is recommended to use custom initialization logic
* When using HasData, you must supply the primary key of the entity – be carefull!

#### Seed Database – By Creating a Custom Seeder Class

1. Create the custom Seeder class and method:

public class AppSeeder

{

private readonly ApplicaitonDbContext \_context;

private readonly IHostingEnvironment \_hosting;

public AppSeeder(ApplicaitonDbContext context,

IHostingEnvironment hosting)

{

\_context = context;

\_hosting = hosting;

}

public void Seed()

{

// Make sure the database exists before starting to seed it:

\_context.Database.EnsureCreated();

if (!\_context.Products.Any())

{

// we don't have any products - create sample data.

var products = new List<Product>()

{

…

}

\_context.Products.AddRange(products);

… additional initializaitons…

\_context.SaveChanges();

}

}

}

1. Add it as a service to Startup.ConfigServices:  
   services.AddTransient<AppSeeder>();
2. Create a RunSeeder private method in Program class that will call the Seeder.Seed():

private static void RunSeeding(IWebHost host)

{

var scopeFactory =

host.Services.GetService<IServiceScopeFactory>();

using (var scope = scopeFactory.CreateScope())

{

var seeder = scope.ServiceProvider.GetService<AppSeeder>();

seeder.Seed();

}

}

1. Call RunSeeding() from main before running the host:

// Build the web host:  
var host = CreateWebHostBuilder(args).Build();

**RunSeeding(host);**

host.Run();

#### Query the Database

In order to query the database in our code we need to connect our class to the ApplicationDbContext. For every class that needs access to the database:

1. Add the following private member:

private readonly ApplicationDbContext \_context;

1. Add a constructor that receives the ApplicationDbContext and initialize the internal member from it:

public AppController(ApplicationDbContext context)

{

context = context;

}

1. Make sure to dispose \_context in the Dispose function:

#### Eager Loading and Reference Loops

* In order to do eager loading of internal references:

\_context.Orders  
 .**Include**(o => o.Item)  
 **.ThenInclude**(I => i.Product);

Like in ASP.Net

* + To include different references of the same object – we can use multiple .Include clauses. One for each property.
  + To include references of the referenced object – we can use .Include and then .ThenInclude like in the example above, where o is the order and i is the item within the order.
* In order to avoid throwing errors on reference loops. For example, order->items->order:  
  Add to services.AddMvc() in Startup.ConfigureServices:  
  .AddJsonOptions(opt => opt.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.RefereLoopHandling.Ignore);
  + Error is the default which will throw an error when it encounter the self-referencing object (e.g. item.order).
  + Serialize will try to serialize it anyway. If we really do have a self-referencing object – this will lead to endless loop.
  + Ignore will ignore the self-referencing object and will not serialize it. In our example: item will return all fields except for the order.

#### Entity Framework command line tools

* > dotnet ef database update – create and update the DB according to the definitions in the code.
* > dotnet ef migrations add <name of migration> - create a new migration (like in ASP.Net)
* > dotnet ef migrations script – will printout the SQL commands to be sent to the database in the current migration (after migrations add).
* > dotnet ef database drop – allow dropping the database completely in order to run all the migrations from scratch. Useful when doing major changes in the database that are not compatible with current data (development only!)

#### The Repository Pattern

Very useful for decoupling our database from our controllers and allowing us to use other databases or even static data for testing!

To implement:

1. In the Data folder: create a new class ApplicationRespository  
   In this class, we’ll implement all our repository data get/set functions like GetProducts, GetProductsByCategory(string category) etc.

**Caution**: Keep ApplicationRepository simple!!! If you define it as generics where you can send any type of data to it, it will become a nightmare to test it….

1. Extract an interface from this new ApplicationRepository.  
   This will allow us to easily mock the database or use a test database for testing.
2. Add the new IApplicationRepository as a scoped service (since it should exist in the same scope as the DbContext) to the Startup.ConfigureServices:

services.AddScoped<IApplicationRepository,

ApplicationRepository>();

### Identity Core

* Supports more types of authentications than in traditional ASP.Net like cookies, OAuth2 etc.
* It’s pluggable:
  + Have complete control over User Entities
  + Support non-relational identities
  + Support LDAP/AD stores

#### Connect to Identity Core

1. Create a new Data\Entities\<AppUser> class to hold the users data.
   1. It should derive from IdentityUser.
   2. You can add any additional fields to it (e.g. FirstName, LastName etc)
2. In your AppContext class, derive from IdentityDbContext<AppUser> instead of from DbContext.
3. Create a new migration to add the new Identity tables to the DB

#### Seed the Default User(s)

1. Add UserManager<AppUser> to your Seeder class and initialize it in the ctor.
2. In Seed(), check for the existence of the user(s) and create them if they don’t exist. Note: creating the users us an async operation so you’ll have to:
   1. Change Seed() to be async Task
   2. await searching for and creating the user

public async Task SeedAsync()

{

…

StoreUser user =

await \_userManager.FindByEmailAsync("some.email");

if (user == null)

{

user = new StoreUser()

{

FirstName = "Ad",

LastName = "Min",

Email = " some.email ",

UserName = " some.email "

};

var result =

await \_userManager.CreateAsync(user,

"P@ssw0rd!");

if(result != IdentityResult.Success)

{

throw new InvalidOperationException(

"Could not create default user in Seeder");

}

}

…

1. wait for the new asynchronous seeding function in RunSeeding:  
   seeder.SeedAsync().Wait();

Note: this will cause a small delay in the startup of the app but not in its run-time performance.

#### Add Roles

To support custom roles (e.g. admin etc), we can derive a new entity from IdentityRole like we’ve done with the IdentityUser

#### Configure Identity

* Add the identity service to ConfigureServices:

services.AddIdentity<AppUser, IdentityRole>( cfg =>

{

cfg.User.RequireUniqueEmail = true;

})

// Tells the FW where to get the identity data from.

// Some projects have different DBContext for data and for identity.

.AddEntityFrameworkStores<DutchContext>();

Note: This will add the default which is cookies-based security. If you have higher security requirements, you will need to add it with the services.AddAuthentication() call. For example:

* Add:

app.UseAuthentication();

In Configure(), before adding the MVC.

#### User

Once we connected the Identity framework in ConfigureServices, we have access to the ‘User’ object that contain the details on the current user.

It allows us to query their login status and their details for logged in users in both code and razor pages. For example:

* @if (!User.Identity.IsAuthenticated)
* if (this.User.Identity.IsAuthenticated)
* var userName = User.Identity.Name;

Note: User.Identity is **just the list of Claims** associated with the current user and not the actual User in our data base. This means that:

* If we need a specific user attribute to be included in the User.Identity, we should add it to the Claims when creating the token.
* If we need to access the actual user in out database (for example, for connecting them to the order), we need to use the UserManager to find them:

private readonly UserManager<StoreUser> \_userManager;

….

var currentUser = await \_userManager.FindByNameAsync(User.Identity.Name);

#### Implement Login and Logout

1. Add a new AccountController : Controller class that will handle login and logout actions.
   1. Add ILogger member for errors loggings.
   2. Add SignInManager<AppUser> member to handle login/logout attempts.
2. Add a new LoginViewModel class that contain the data required by the login page
3. Add the new Login page that uses the LoginViewModel properties.
4. Add Login() action that displays the page and add the link from the main page that opens it.
5. Add Login(LoginViewModel loginViewModel) action that authenticate the login:

HttpPost]

public async Task<IActionResult>

Login(LoginViewModel loginViewModel)

{

if(ModelState.IsValid)

{

var result = await

\_signInManager.PasswordSignInAsync(

loginViewModel.UserName,

loginViewModel.Password,

loginViewModel.RememberMe,

false);

if (result.Succeeded)

{

if (Request.Query.Keys.Contains(

"ReturnUrl"))

{

Redirect(

Request.Query["ReturnUrl"].First());

}

else

{

RedirectToAction("Shop", "App");

}

}

}

// This will add this error to the model, to be

// displayed in the

// <asp-validation-summary="ModelOnly">

// in the login view.

ModelState.AddModelError("", "Failed to login");

return View();

}

1. Add logout action and link:

[HttpGet]

public async Task<IActionResult> Logout()

{

await \_signInManager.SignOutAsync();

return RedirectToAction("Index", "App");

}

#### Action Authorization

* Add [Authorize] to any controller/action that’s for logged-in users only
* If you need to specify a different authentication scheme from the default cookies, you can add it to the [Authorize] annotation:

[Authorize(AuthenticationSchemes =

JwtBearerDefaults.AuthenticationScheme)]

### AutoMapper

Like with ASP.Net we can use AutoMapper and AutoMapper.Extensions.Microsoft.DependencyInjection to map between our controllers and our ViewModels.

#### To install:

1. In Manage Nuget Packages, search and install AutoMapper and AutoMapper.Extensions.Microsoft.DependencyInjection
2. Create a new AppMappingProfile.cs under Data:

public class DutchMappingProfile **: Profile**

{

public DutchMappingProfile()

{

**CreateMap**<Order, OrderViewModel>()

**.ForMember**(o => o.OrderId,

ex => ex.MapFrom(o => o.Id))

**.ReverseMap**();

}

}

Notes:

* + .ForMember(<lambda expression for the destination>,  
     <member options lambda>
  + .ReverseMap() will create the mapping in both directions.

1. In Strartup.ConfigureServices, add:  
   services.AddAutoMapper();

#### To Use:

1. In any controller that requires a new mapper, add an ): IMapper \_mapper readonly private field which will be initialized in the ctor (like the other services) to be used for mapping.
2. Mapping:

\_mapper.Map<fromType,toType>(objectToMap);

Or, for all collection types. For example:

\_mapper.Map<IEnumerable<fromType>,  
 IEnumberable<toType>>(objectCollectionToMap);

#### ViewModel Sub-Properties

* Like in the Asp.Net, every ViewModel type should only include ViewModel items!! For Example:

public class OrderViewModel

{

…

public ICollection<OrderItemViewModel> Items { get; set;}

}

* AutoMapper has a special feature – if you prefix the properties of a sub-item with the sub-item name, it will automatically know where to find these properties and will auto map them for you.  
  for example: OrderItem contains the Product reference. So in OrderItemViewModel, we might need to include a new ProductViewModel member. However, if we only need some of the fields of the Product for displaying only (no need to map the full object) – we can instead pre-fix all the fields we need with ‘Product’:

public class OrderItemViewModel

{

public int Id { get; set; }

…

[Required]

public int ProductId { get; set; }

public string ProductCategory { get; set; }

public string ProductArtId { get; set; }

…

}

So, Automapper will know to map the Product<field> fields to Product.<field> automatically.

### API Controllers

In .Net Core, the WebAPI is integrated into the ASP.Net Core and we don’t need to derive it from different library/folders.

#### Create an API Contoller

Route("api/[Controller]")]

public class ProductsController : Controller

{

private readonly IDutchRepository \_repository;

private readonly ILogger<ProductsController> \_logger;

public ProductsController(IDutchRepository repository,

ILogger<ProductsController> logger)

{

\_repository = repository;

\_logger = logger;

}

[HttpGet]

public IActionResult Get()

{

var products = \_repository.GetProducts();

if (products != null)

{

return Ok(products);

}

else

{

return BadRequest("Failed to find products");

}

}

}

#### API Contoller Actions

* The default route for the controller will be the one specified in the [Route()] attribute of the class. This is the root of the controller.  
  The [Controller] in the route will refer to the name of the controller. For example: Product for ProductController, Item for ItemController etc.
* The route can also include parmeter mapping using {}. For example:

**[Route("api/orders/{orderid}/items")]**

public class OrderItemsController : Controller

{

…

[HttpGet]

public IActionResult GetItems(int orderId)

{…}

}

* For every action, we can specify special route from the controller’s root route in the annotation of the action:  
  [HttpGet(“{id:int}”)]

If we don’t specify the type – the default is string.

* Post:
  + Buy default, the [HttpPost] action will expect to receive the data through query string (?id=1234 etc). If we want to receive the data thorugh the query’s body (which we usually would), we can specify this be adding [FromBody] annotation to the received data. For example:  
    [HttpPost]  
    public IActionResult Post(**[FromBody]**Order model){..}
  + AddEntity: since to add a new object into the DB, we

#### API Controller Returned Type

* The collection/item itself – problem: if there was an error and the Repository layer returned null, we can return null to the users but we don’t have a good way to return the reason for the error.
* JsonResult:

public JsonResult Get()  
{

try

{

Return Json(\_repository.GetAllProducts());

}

Catch (Exception ex)

{

\_logger.LogError($”Failed to get products: {ex}”);

Return Json(“Bad Request”);

}

}

problem: this is very rigid, not allowing the MVC system to return different kinds of data like it does in Views.

* IActionResult:

public IActionResult Get()  
{

try

{

Return Ok(\_repository.GetAllProducts());

}

Catch (Exception ex)

{

\_logger.LogError($”Failed to get products: {ex}”);

Return BadRequest(“Failed to get prodcuts”);

}

}

Just like in the views’ controllers. This allows us maximum flexibility and is the preferred method.

#### Association Controller

A controller that allow the user to access the different collections at different levels. So for example, not only to get a full order (/api/orders/1) but also to be able just to look at the list of items in this order and change it (/api/orders/1/items).

To create an Association Controller:

1. Add a new controller (e.g. OrderItemsController)
2. Add the route, including the parameters to the class:  
   [Route("api/orders/{orderid}/items")]
3. For every action, include the class parameter (e.g. orderId) as well as any action-specific parameter e.g. [HttpGet(“{itemId}”)]

#### Identity with API

**Can we just use cookies?**

Can we use the same cookie that our web views use for our API?  
Yes, but:

* + It means that the user must use the web-view part of your site in order to login before using the web API.
  + It’s less secure:
    - Since the cookies can keep the user logged in for weeks and months, it means that they can be grabbed and miss-used.
    - Since calling the web API can change data potentially so the risk is much greater than when using the views.
  + If you don’t have a choice, you can still use cookies since there is some security there but they are not the best option,
  + Better options are: Open ID, OAuth2 or Jason Web Token (JWT) Tokens.
  + Your choice should depend on your security requirements and not on the effort level!

**Add Identity with JWT to Your API:**

This authentication method works in the following way:

* + 1. The user will need to call an action to create a token and supply their login details
    2. If the action succeeds, the action will return a token
    3. The user can now send this token as part of their http requests’ header

**To Implement:**

* + 1. Add the Tokens Key, Issuer and Audience to your configuration file.  
       These value will need to be assigned by the security team and changes if/when required. For example, in config.json:

"Tokens": {

"Key": "adfabsu vusn waoe qp32-83jf;a jfd ldsfdsfal",

"Issuer": "localhost",

"Audience": "users"

}

* + 1. Add the required authentication method in ConfigureServices:

services.AddAuthentication()

.AddCookie()

.AddJwtBearer(cfg =>

{

cfg.TokenValidationParameters = new

TokenValidationParameters()

{

ValidIssuer = \_config["Tokens:Issuer"],

ValidAudience = \_config["Tokens:Audience"],

IssuerSigningKey = new

SymmetricSecurityKey(

Encoding.UTF8.GetBytes(

\_config["Tokens:Key"]))

};

});

* + 1. Add an action in the AccountController to create a token to use in API calls; see in AspDotNetCoreFromScratch.AccountController.CreateToken

**To Use:**

* + In Code:

Every controller/action that requires authentication should include the data annotation:

[Authorize(AuthenticationSchemes =

JwtBearerDefaults.AuthenticationScheme)]

* + By Users:
    1. Get the token by posting the login details to the account/CreateToken action.
    2. If the action succeeds, the action will return a token
    3. The user can now send this token as part of their http requests’ header by adding the following Key and value to the header:
       - Key: Authorization
       - Value: Bearer <the received token>

#### Building Public API

When building public API, you need to document it well. Swagger is a tool for taking the code and building the documentation from it.

To build a public API and be able to use the automatic tools for documentation, we’ll need to do the following changes:

* Make sure our application supports ASP.Net Core 2.1 features by adding to the services.AddMvc() call in Startup.ConfigureServices:  
  .SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);
* Our API Controller needs to inherit from ControllerBase instead of Controller
* We should add the following annotations to the class:
  + [ApiController]
  + [Produces(“application/json”)] – this is the most common returned type but we can set it to other things as well.
* Our Actions should return the generic-concrete type ActionResult<returned type> instead of the interface type IActionResult
* For each action, we can (although I don’t like it – horrible dry) add:  
  **[ProducesResponseType**(<code>)] – for each possible response code that the function returns. For example:   
  [ProducesResponseType(200)]

### TypeScript

* Add any customer-side typescript files under wwwroot/ts
* In the root of the project, add tsconfig.json file that will hold the typescript compiler configurations:

{

"include": [ "wwwroot/ts/\*\*/\*.ts" ],

"compilerOptions": {

// to support accessors:

"target": "es5",

// to create the source map needed 4 debugging

"sourceMap": true

}

}

#### Debugging TypeScript in the Browser

If you used the “sourceMap” option in the tsconfig.json, you should be able to see the .ts files and debug them in the browser.

### Add NuGet Packages

For server-side packages:

Solution -> right click -> Manage NuGet Packages

### Add Javascript Packages

1. Add NPM support to allow installing and managing JS packages:  
   solution -> right click -> Add new Item -> c# -> ASP.Net Core -> Web -> npm Configuration File

This will add the package.json at the root directory of your project where you can manage JS dependencies.

1. To enable serving the javascripts from node\_modules, add the NuGet package OdeToCode.UseNodeModules.   
   Then, add to the Startup.Configure method:  
   app.UseNodeModules(env);
2. To install a new package in your solution, either:
   1. In the package.json, add the required packages like you would normally. Visual Studio will automatically download and put the packages into node\_modules, usually under the <package name>/dist directory.
   2. Open command line in your solution folder and use the   
      > npm install command as usual.
3. To use the package in the html, add the script like you would with raw html with the source pointing to the required .js file.

### Tag Helpers

<https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers>

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. It is similar to ASP.Net Html Helpers (see HTML Helpers), but unlike HTML Helpers, **they don’t call C# function that generate HTML tags. Instead, they attach custom attributes to the tags that is used by Razor to render the html**.

Example – with HTML Helper:

@Html.Label("FirstName", "First Name:", new {@class="caption"})

With HTML Tags:

<label class="caption" asp-for="FirstName"></label>

* + Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper.
  + Where **Tag Helpers** are available, they are **more readable and provide better intelliSense** then the parallel HTML Helpers.

**Add Support for Tag Helpers**

If not included already:

Add Views/\_ViewImports.cshtml file and inside it add:

@using <project name space>.Controllers

@addTagHelper "\*, Microsoft.AspNetCore.Mvc.TagHelpers"

This will add support for the standard Microsoft tags helpers.

We can also implement additional tags helpers and add them here.

**Common Tag Helpers:**

* + asp-controller
  + asp-action
  + asp-for – for a label – will set the ‘for’ attribute. For input field – will set both the ‘id’ and the ‘name’ attributes.
  + asp-validation-summary – to show all validation errors.
    - “All” – all validation errors in all the fields.
    - “ModelOnly” – only validation errors on the model as a whole (not per-field).
  + Asp-validation-for – per field. Validate each field as the user enter data into it.  
    NOTE: You need to :
    - install “jquery-validation” and “jquery-validation-unobtrusive” js packets for this to work.
    - Include the .min.js files (first validation and then validation-unobtrusive) in the required view, using the @section directive (see Razor commands).

### Razor Pages

When we need a simple cshtml page that isn’t connected to a controller, for example – for displaying a generic error page - we can use Razor Pages:

* + Add a .cshtml file under Pages folder
  + The page must start with @page to mark that this is a Razor page, not connected to the view-controller scheme.
  + If you want to have the same layout in these pages as the rest of your application, you need to copy the \_ViewStart.cshtml to this folder as well ☹

### Services

* + Live under Services folder
  + Called <Name>Service.cs
  + To add a service:
    - Create the new service.cs class
    - Do you need to extract an interface from this service (do you expect different providers that you’ll need to adapt to)?
    - In Startup.ConfigureServices, add:  
      services.AddTransient<IMailService, NullMailService>();
      * services.AddTransient – for services that don’t have their own data and usually just do things (e.g. mailservice)
      * services.AddScoped – for services that are more expensive to initialize and that are kept for the life of the request
      * services.AddSingleton – for services that are initialized once and kept for the life of the application

Notes: adding a service has two options (the examples below works the same for all kinds of services scopes):

// if the service doesn’t have an interface –

// just one concrete class:  
services.AddTransient<ConcreteService>();

// if the service as an interface defined on it –  
// the call will take the interface and the concrete  
// class to use for initializing the service with:

Services.AddTransient<IServiceInterface,ConcreteService>();

* + - In the controller that needs the service, inject the service by adding it to the controller’s constructor:

public AppController(IMailService mailService){..}

Note: the controller doesn’t know or care which type of IMailService it receives, as long at it support the expected interface. The dependency-injection layer is the one that knows to send the NullMailService and how to construct it for sending to the controller.

### Useful Project Configurations

* In Debug->App URL – you can change the port here.

### Useful .Net Core Packages

* Newtonsoft.Json – Json functions for example:   
  using Newtonsoft.Json;  
  …  
  Console.WriteLIne(JsonConvert.SerializeObject(objectToPrint));
* Environment – Development/Stagging/Production:  
  solution -> properties -> Debug -> Environment variables -> ASPNETCORE\_ENVIRONMENT  
  this variable can control the env.IsDevelopment/Stagging/Production(); call result (although I except there to be a better way to control it???)
* AutoMapper and AutoMapper.Extensions.Microsoft.DependencyInjection – for auto-mapping between entities and ViewModels.

### Logging

#### Add Logging

To add logging from a class, add the ILogger<classType> to the class’ constructor and connect it to an internal member that will be used for logging:

public class **NullMailService** : IMailService

{

private readonly **ILogger<NullMailService>** \_logger;

public NullMailService(**ILogger<NullMailService> logger**)

{

\_logger = logger;

}

public void SendMessage()

{

**\_logger.LogInformation**(“Send Message Called”);

}

}

#### Set Logging Level

To control the level of logging, add the following to your config file:

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning"

}

}

* The “Microsoft” log level means – all logging that comes from classes starting with “Microsoft”

### Run the App from Command Line

> set ASPNETCORE\_ENVIRONMENT=Development // if required

> dotnet run

### Deployment

#### Environment

In your cshtml files, you can use the

<environment names=””>… </environment> tag in order to include certain tags conditionally on the environment. For example:

<**environment** **names**="Development">

<script src="~/node\_modules/jquery/dist/jquery.js"></script>

<script src="~/node\_modules/bootstrap/dist/js/bootstrap.bundle.js">  
 </script>

<script src="~/js/index.js"></script>

</**environment**>

<**environment** **names**="Staging,Production">

<**script** **src**="//ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.js"

**asp-fallback-src**="~/node\_modules/jquery/dist/jquery.min.js"

**asp-fallback-test**="window.jQuery"></**script**>

<**script**   
 **src**="//ajax.aspnetcdn.com/ajax/bootstrap/4.3.1/bootstrap.bundle.js"  
 **asp-fallback-src**=  
 "~/node\_modules/bootstrap/dist/js/bootstrap.bundle.min.js"  
 **asp-fallback-test**=  
 "window.jQuery && window.jQuery.fn &&   
 window.jQuery.fn.modal"></**script**>

<script src="~/node\_modules/bootstrap/dist/js/bootstrap.bundle.min.js">  
 </script>

<**script** **src**="~/dist/dutchtreat.min.js" **asp-append-version**="true">  
 </**script**>

</**environment**>

Notes:

* + **asp-append-version**="true" will add a version after the file name so that we publish a new version, it will have a different number than the previously released one so that the browser will have to get it again and not use the cached version.
  + To save time/resources in getting common scripts like jQuery and bootstrap to the user’s computer:  
    <**script** **src**="<path for the script on CDN e.g. //ajax.aspnetcdn.com..  
    **asp-fallback-src**="script path on server – if couldn’t find/reach CDN"  
    **asp-fallback-test**="a Boolean expression for testing if script already available on the client’s computer"></**script**>
  + You can change the project’s environment name when you right click the project -> properties -> Debug -> Environment variables -> ENVIRONMENT and set the value to Development/Staging/Production

#### Gulp

Gulp is a tool for automating our build process including minimizing our css and js files.

Note: as of 26.4.2019 – Gulp doesn’t support all ES5 features. Specifically, it doesn’t support `` strings and ‘let’.

In order to set up gulp:

1. Add the following to the "devDependencies” in package.json:
   1. gulp
   2. gulp-uglify (mimify)
   3. gulp-concat (combine a lot of files into one file)
2. At the root of your project add a new gulpfile.js file. For example:

var gulp = require("gulp");

var uglify = require("gulp-uglify");

var concat = require("gulp-concat");

function minify() {

return gulp.src("wwwroot/js/\*\*/\*.js")

.pipe(uglify())

.pipe(concat("dutchtreat.min.js"))

.pipe(gulp.dest("wwwroot/dist"));

};

function defaultTask(cb) {

minify();

cb();

}

exports.default = defaultTask;

**Run from command line:**

To run gulp, from the command line:  
> gulp

**Run as part of the Visual Studio build:**

1. Right click on ‘gulpfile.js’->Task Runner Explorer
2. Right click on the gulp task you want to run -> Binding -> choose which binding you want to set.   
   This will add a binding header to the gulpfile.js and will run automatically with your builds.

#### Set up Deployment Scripts

* + - 1. Add the following to the bottom of your project’s csproj file (right click project -> Edit csproj file):

<Target Name="MyPublishScripts" BeforeTargets="BeforePublish">

<Exec Command="npm install" />

<Exec Command="gulp" />

<Exec Command="ng build" />

</Target>

* + Name – is the name of this batch of commands.
  + BeforeTargets – the stage (.Net hard-coded) when to run it.
  + A list of Exec commands to run.
  + In order for this deployment method to work for me, I had to:
    1. Remove the gulp command. Since the gulpfile already runs automatically on every build, I verified that it still created and copied the minified application to the correct folder in the target.

#### Publish to a Directory

1. Right click project -> publish -> folder and press ‘Publish’
2. Open a command line in the target folder and run: npm install manually on the target folder for the node\_modules to be installed.

In order to run – browse to the folder on command line and run:

> dotnet <dll file>

#### Publish to Azure

1. In the publish type, choose ‘App Service’ -> Azure App
2. Click publish. VS will run the same build process as with the file only instead of copying the files to the output directory, it will upload them to your Azure account and will immediately try to run it from Azure.

\*\* This didn’t work for me. I’ll need to investigate this in the future if/when I need to…

#### Publish to IIS

On my PC:

1. Open IIS Manager
   1. Under Sites -> Create new web site.
   2. Application pool: CORE
   3. Create a folder for this deployment
2. In visual studio:
   1. Reopen visual studio in admin mode
   2. Create a new Publish Profile -> IIS -> Publish Method: Web deploy

#### Publish through Command Line

> dotnet publish -o “<output directory>”

It will run the same build and deploy like the Visual Studio IDE one.

#### Publish with the Run Time

If we need to publish our application with the .Net Core Runtime so that it won’t be dependent on any operating system we can do the following:

1. Add the required RuntimeIdentifier tag to the <PropertyGroup> tag in the project’s .csproj file. For example:

<RuntimeIdentifier>win10-x64</RuntimeIdentifier>

1. When building from the command line, add the - - self-contained flag to the build. This will include all required .dll files in the output directory. For example:

> dotnet publish -o “<output directory>” –-self-contained

This will also create the application as an .exe file.

We can also set <RuntimeIdentifiers> to support multiple runtimes and then in the command line we can decide which runtime we’re publishing for. For example:

<RuntimeIdentifiers>

win10-x64,OSX.10.10-x64

</RuntimeIdentifiers>

> dotnet publish -o “<output directory>” --runtime osx.10.10-x64

Note: this didn’t work for me but I’m not spending time on this now.

**Runtime Identifier**

If our application uses and rely on platform or runtime-specific features, we might need to publish our site with its RID. RID is short for Runtime IDentifier. RID values are used to identify target platforms where the application runs. They're used by .NET packages to represent platform-specific assets in NuGet packages. The following values are examples of RIDs: linux-x64, ubuntu.14.04-x64, win7-x64, or osx.10.12-x64. For the packages with native dependencies, the RID designates on which platforms the package can be restored.

Examples of where we might want to use this:

* + When we publish different parts of our site with different runtime versions.
  + When our site depends on platform-specific APIs
  + When we deploy on the cloud in a virtual machine and don’t want to have to setup the .Net Core environment on the virtual machine.

## Angular App in .Net Core

1. Create your new Angular app on a different temporary directory than your project using the angular command line tool ng.
2. To copy the new project into your ASP.Net Core project:
   1. Close all files in your project and make sure that your Angular project is not running.
   2. In your ASP.Net Core project: create a new folder at your project’s root. Let’s call it ClientApp
   3. Copy the following files from your angular-app root directory to your ASP.Net Core app:
      * angular.json
      * tsconfig.json
   4. Copy everything under your angular-app/src folder to your new ClientApp folder.
   5. Add all the ‘dependencies’ and ‘devDependencies’ from the angular-app package.json to our ASP.Net package.json.
   6. We can now delete the angular-app folder.
   7. In tsconfig.json:
      * add  
        “exclude”: [“./node\_modules/”]  
        so that the TS compiler will not try to re-compile node modules.
      * Change "outDir" to be:   
        "./wwwroot/clientapp/out-tsc",
      * in the solution explorer -> Right-click on tsconfig.json file -> properties -> Build Action -> None. This will prevent VS from trying to compile the Angular TS files, causing compilation errors.
   8. In angular.json:
      * set “projects”.”angular-app”.”root” and “sourceRoot” to “ClientApp”
      * set build.options.outputPath to “wwwroot/clientapp/dist”
      * replace all file name paths from src/ to ClientApp/
3. install all the required packets:

> npm install

1. Build the angular app:  
   > ng build
2. To integrate the angular-app into our project:

For every view (cshtml) that you want to use an angular component:

* + - Include the following scripts, in this order:

@section Scripts {

<script src="~/clientapp/dist/runtime.js"></script>

<script src="~/clientapp/dist/polyfills.js"></script>

<script src="~/clientapp/dist/styles.js"></script>

<script src="~/clientapp/dist/vendor.js"></script>

<script src="~/clientapp/dist/main.js"></script>

}

* + - Add the tags to be replaced, according to the component’s ‘selector’.

## Troubleshooting

### Views/CSS

1. Problem: The page doesn’t refresh after I changed the view or CSS
   1. Solution: Press Ctrl+F5 in the browser to force a refresh.

# Azure

Azure Portal

# Coding Conventions

<https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions#language-guidelines>

## Linters

1. FxCop – integrated into VS as “Code Analysis”
2. Stylecop

## Coding Style

Microsoft’s [All In One Code framework](http://1code.codeplex.com) has a Coding Style Guideline document. The Microsoft All-In-One Code Framework is a free, centralized code sample library provided by the Microsoft Community team. It has typical code samples for all Microsoft development technologies, and a code style guideline document with that. Thanks to Kevin for pointing out this guideline document with All In One Code Framework (See the comments)

* [Download All In One Code Framework Guideline Document](http://1code.codeplex.com/wikipage?title=All-In-One%20Code%20Framework%20Coding%20Standards&referringTitle=Documentation)

- See more at: http://www.amazedsaint.com/2010/11/top-6-coding-standards-guideline.html#sthash.O2o7ZyOH.dpuf

The most recommended C# coding standard I found is IDesign C# Coding Standard

* 1. DO prefix interface names with the letter I, to indicate that the type is an interface.For example, IComponent
  2. Always use C# predefined types rather than the aliases in the System namespace. For example:
     1. Use object and not Object
     2. Use string and not String
     3. Use int and not Int32

|  |  |  |  |
| --- | --- | --- | --- |
| Identifier | Casing | Naming Structure | Example |
| **Class, Structure** | PascalCasing | Noun | public class ComplexNumber {...}  public struct ComplextStruct {...} |
| **Namespace** | PascalCasing | Noun  🗷 **Do not** use the same name for a namespace and a type in that namespace. | namespace Microsoft.Sample.Windows7 |
| **Enumeration** | PascalCasing | Noun  🗹 **Do** name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases. | [Flags]  public enum ConsoleModifiers  { Alt, Control } |
| **Method** | PascalCasing | Verb or Verb phrase | public void Print() {...}  public void ProcessItem() {...} |
| **Public Property** | PascalCasing | Noun or Adjective  🗹 **Do** name collection proprieties with a plural phrase describing the items in the collection, as opposed to a singular phrase followed by “List” or “Collection”.  🗹 **Do** name Boolean proprieties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with “Is,” “Can,” or “Has” but only where it adds value. | public string CustomerName  public ItemCollection Items  public bool CanRead |
| **Non-public Field** | camelCasing or \_camelCasing | Noun or Adjective.  🗹 **Do** be consistent in a code sample when you use the '\_' prefix. | private string name;  private string \_name; |
| **Event** | PascalCasing | Verb or Verb phrase  🗹 **Do** give events names with a concept of before and after, using the present and past tense.  🗷 **Do not** use “Before” or “After” prefixes or postfixes to indicate pre and post events. | // A close event that is raised after the window is closed.  public event WindowClosed  // A close event that is raised before a window is closed.  public event WindowClosing |
| **Delegate** | PascalCasing | 🗹 **Do** add the suffix ‘EventHandler’ to names of delegates that are used in events.  🗹 **Do** add the suffix ‘Callback’ to names of delegates other than those used as event handlers.  🗷 **Do not** add the suffix “Delegate” to a delegate. | public delegate WindowClosedEventHandler |
| **Interface** | PascalCasing  ‘I’ prefix | Noun | public interface IDictionary |
| **Constant** | PascalCasing for publicly visible;  camelCasing for internally visible;  All capital only for abbreviation of one or two chars long. | Noun | public const string MessageText = "A";  private const string messageText = "B";  public const double PI = 3.14159...; |
| **Parameter, Variable** | camelCasing | Noun | int customerID; |
| **Generic Type Parameter** | PascalCasing  ‘T’ prefix | Noun  🗹 **Do** name generic type parameters with descriptive names, unless a single-letter name is completely self-explanatory and a descriptive name would not add value.  🗹 **Do** prefix descriptive type parameter names with T.  🗹 **You should** using T as the type parameter name for types with one single-letter type parameter. | T, TItem, TPolicy |
| **Resource** | PascalCasing | Noun  🗹 **Do** provide descriptive rather than short identifiers. Keep them concise where possible, but do not sacrifice readability for space.  🗹 **Do** use only alphanumeric characters and underscores in naming resources. | ArgumentExceptionInvalidName |

# Unit Testing

## NUnit

### Install

>> Install-package NUnit  
Tools->Extensions and Updates->NUnit Test Adapter

### Project Setup

1. Create new unit tests project:  
   New->Project->Class Library-><unit test project>  
   Convention: <project\_under\_test>.UnitTests
2. In the unit test project->References->add reference to the project under test.
3. Conventions:
   * Name of Fixture: <class under test>Tests
   * name of parameter to hold the System Under Test:  
     var sut = new <SystemUnderTest>
   * Test name: If<condition>Then<result>

In order to use Nunit:

1. Make sure that your project’s output is define as class library
2. Right click on your solution in Visual Studio -> Add | New project…
3. Click on Visual Visual C# Unit Test Project.
4. Enter the name of your test project and click OK (UT\_name of project for testing).
5. On the new UT project:
   1. Tools -> NuGet Package Manager -> Manage NuGet Packages for Solution -> Browse -> choose Nunit and NUnitTestAdapter
   2. Manage which projects uses it (UT only) and install if required.
   3. Right click -> Add -> Reference to the project under test.
6. On the solution-> right click -> properties -> dependencies. Define the unit test project as dependent on the project under test.

### Code

using Nunit.Framework;

namespace UT\_<NameOfTheUnitTestNameSpace>

{

**[TestFixture]**

public class UT\_<NameOfTheClassOrFeatureToTest>

{

**[Test]** public void UT\_<When**\_**StateUnderTest\_Should\_ExpectedBehavior>()

{

… test

Assert.AreEqual(<expected result>,   
 <call the functionality under test>);

Assert.That(<result>,Is.EqualTo(<expected result>);  
Assert.That(<result>,Is.Not.EqualTo(<forbidden result>);  
Assert.That(<result>,Is.Null);

}

}

}

### Check Exceptions

Assert.Throws<FileNotFoundException>(() => <call the function that throws exception>);

If we want to know if it’s a general/derivative exception:

Assert.Throws(Is.InstanceOf(typeof(Exception)), () => <call the function that throws exception>);

That’s also how we check exceptions in constructor:

Assert.Throws(Is.InstanceOf(typeof(Exception)), () => new MyObject(params..));

## Mocking

Moq is a very popular mocking framework for C#.

### Installation

To install it in .Net Studio:

1. Tools->NuGet Package Manager -> Package Manager Console…  
   PM> Install-Package Moq
2. Right click on project -> Manage NuGet Packages…  
   find Moq and press install
3. Right click on project -> Add Reference -> Browse to Moq installation and choose Moq.dll

### Usage

1. Include Moq:

using Moq;

1. Create a mock object:

var \_mock = **new** Mock<ICustomerView>();

1. Set up the mock behaviour: See *Mock Set up*
2. Inject the mock object into the object under test:

BLCustomer blCustomer = new BLCustomer(\_mock**.Object**);

1. Act: run the code:

blCustomer.DeActivateCustomer();

1. Verify:
   * + - Either with regular unit tests’ Asserts or

### Mock Set up

* Return a specific answer:  
  \_mock.Setup(x => x.GetCustomerTotal(It.IsAny<int>()))

.Returns(25.5);

Will return 25.5 when GetCustomerTotal is called with any integer

* Return set value depending on the values passed to the method:
* Define Property’s value:  
  mockCustomerDl.SetupGet(prop =>

prop.IsActive).Returns(true);

* Set the mock to return a value when a propery’s get is called:

mockCustomerDl.**SetupGet**(prop => prop.IsActive).Returns(true);

Will return true whenever the property IsActive is queried

* Setup mock object property to track values (stub):
* Setup mock object to throw exceptions:
* To verify that a method was called, use the Verify method on the mock object;

mockCustomerRepository.Verify(t =>

t.Add(It.IsAny<Customer>()));

* Verify mock object methods were not called:
* Verify the number of times that a mock object method was called:

\_mockValidator.Verify(x =>

x.IsValid(It.IsAny<string>()),Times.Once);

* Verify the returned value from an internal function call
  1. Whenever

var \_mock = new Mock<IBasketDal>();

Then we need to tell this mock what its going to do, and what it will return

### Mocking the Console

Instead of mocking the console, it is possible to use StringWriter to redirect the system’s output to a string that can be later asserted:

using (StringWriter sw = new StringWriter())  
{

Console.SetOut(sw);

SomeFunctionWithOutput();

Assert.AreEqual(“Expected Output”, sw.ToString();

}

# Full Stack Testing

## xUnit.net

A free, open source unit testing framework for .Net Framework.

### Test Attributes

Attributes that we can put on the test methods:

* [Fact] – run the test as is
* [Theory] – run test with supplied data (in InlineData)
* [InlineData(data)] – supply the data for running the test with the [Theory] attribute.

### Assertion Methods

* Assert.Equal()
* Assert.False()
* Assert.True()
* Assert.Throws()

### Test Execution Lifecycle Management

* Test setup
* Test cleanup
* Shared test context/data/instances

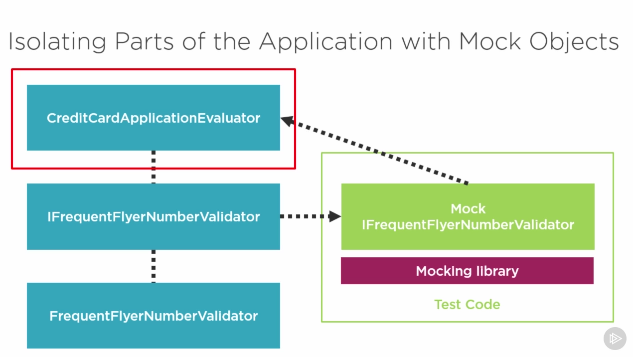
### Testing Model using with xUnit.net

1. Add test folder under the root directory of the solution
2. Right click -> Add New Project -> xUnit Test Project (.NET Core)
3. In the solution explorer: under the test project->dependencies->Add reference to the production project.
4. Add a new test class:

using System;  
using Xunit;  
  
public class <ClassUnderTest>Should{…}

1. Each test should complete the sentence. For example:  
   [Fact]  
   public void AcceptValidName(){…}

### Mocking



### Run Tests

* In Visual studio -> open tests explorer and run from there
* Through command line:
  + testProjectRootDirectory> dotnet test

will discover and run all the tests in the project.

* + >dotnet test ? – for help