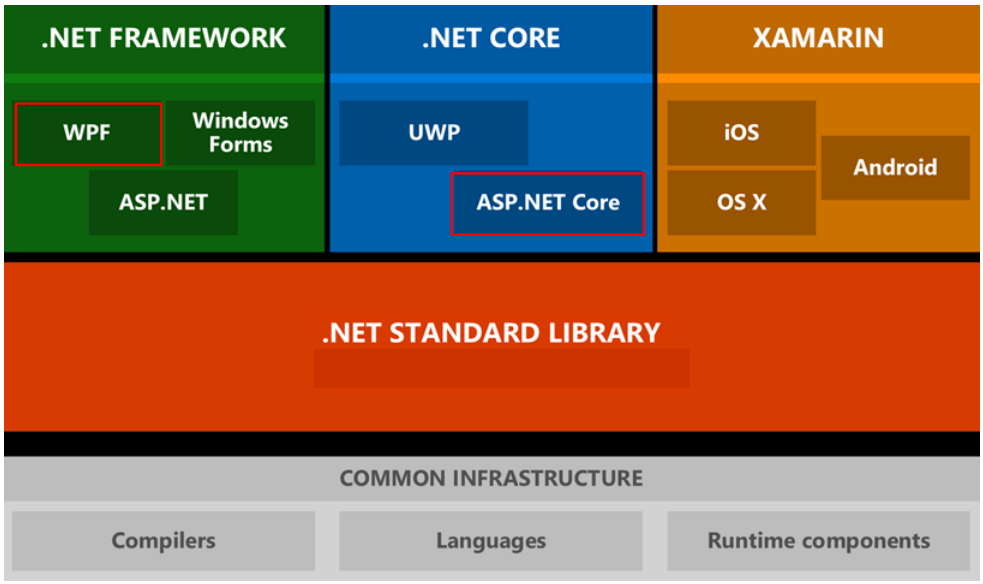
.NET Course

Contents

Introduction to the .NET framework

.NET Standard



.NET Standard is a set of APIs that are implemented by the Base Class Library of a .NET implementation. Code targeting a version of the .NET Standard can run on any .NET implementation which supports that version of the .NET Standard.

Each implementation of .NET includes the following components:

- One or more runtimes. Examples: CLR for .NET Framework, CoreCLR and CoreRT for .NET Core.
- A class library that implements the .NET Standard and may implement additional APIs. Examples: .NET Framework Base Class Library, .NET Core Base Class Library.
- Optionally, one or more application frameworks. Examples: ASP.NET, Windows Forms, and Windows Presentation Foundation (WPF) are included in the .NET Framework.
- Optionally, development tools. Some development tools are shared among multiple implementations.

Framework	Use for	Version
.NET Core	Cross-platform cloud hosted applications	2.0
.NET Framework	Web and Windows Desktop applications	4.6.1

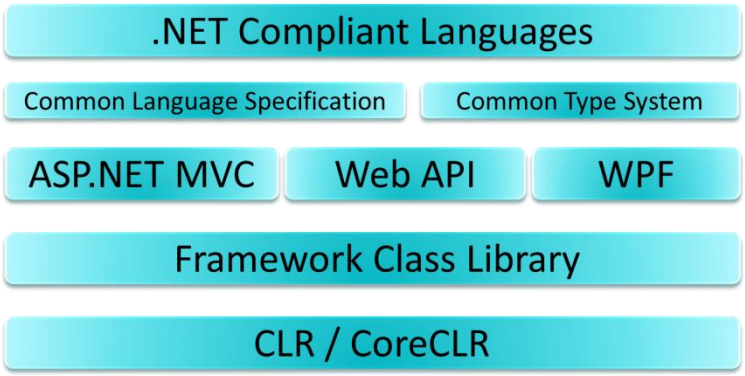
NET Core

.NET Core is a cross-platform implementation of .NET and designed to handle server and cloud workloads at scale. It runs on Windows, macOS and Linux. It implements the .NET Standard, so code that targets the .NET Standard can run on .NET Core. ASP.NET Core runs on .NET Core.

.NET Framework

The .NET Framework is the original .NET implementation, containing Windows-specific APIs, such as WPF. Version 4.7 implements .NET Standard 2.0

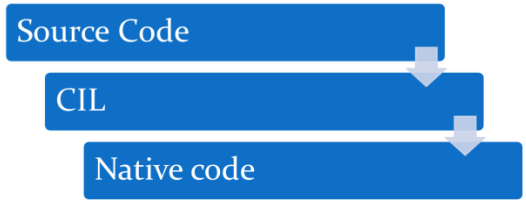
CLR



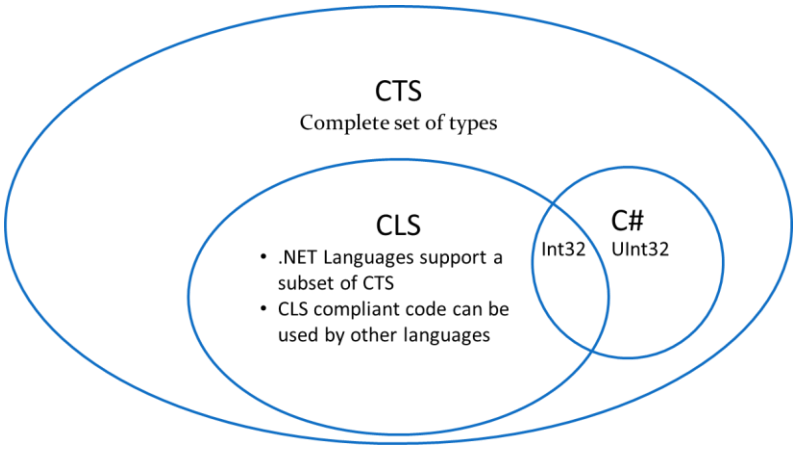
Programs written for the .NET Framework execute in the Common Language Runtime (CLR), a runtime environment that provides services such as security, memory management, and exception handling.

Common Intermediate Language

CIL is a stack-based object-oriented assembly language. At compile time, C# source code is translated into CIL. At runtime, the CLR's Just-In-Time (JIT) compiler translates CIL into native code, which is then executed by the computer's processor.

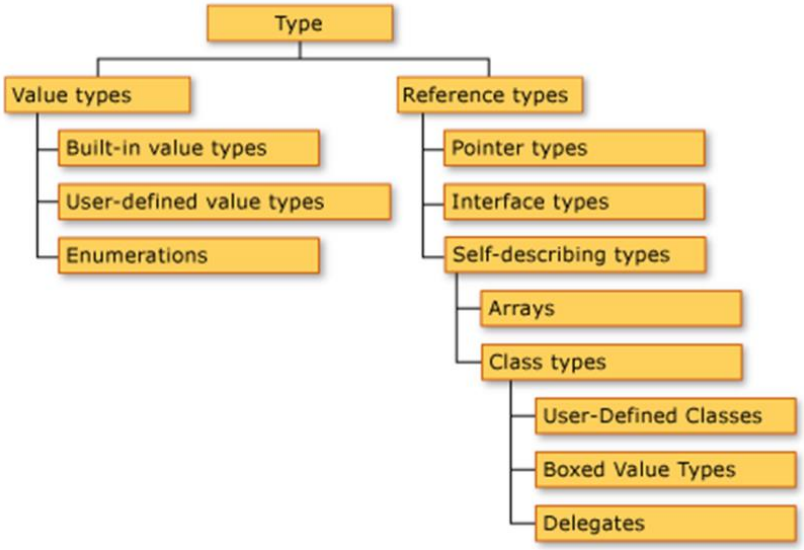


Common Language Specification



To fully interact with other objects written in any language, objects must expose to callers only those features that are common to all languages. This common set of features is defined by the Common Language Specification (CLS). For example UInt32, a 32 bit unsigned integer, is in the CTS and available to C#, but is not in the CLS.

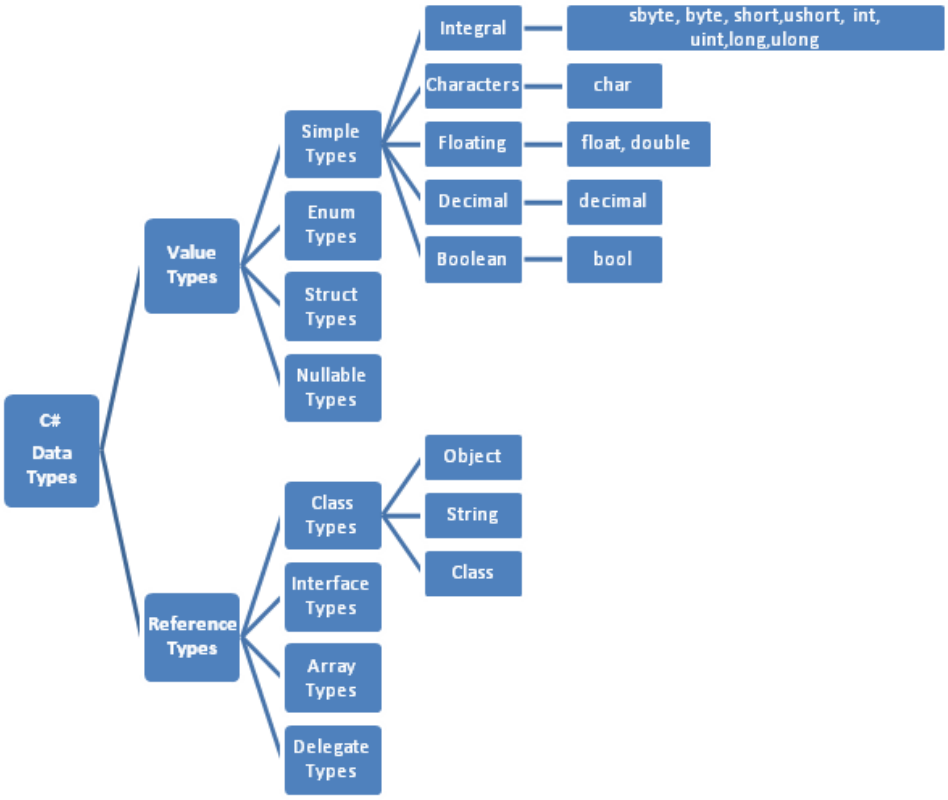
Common Type System



The common type system defines how types are declared, used, and managed. It’s an important part of the runtime's support for cross-language integration.

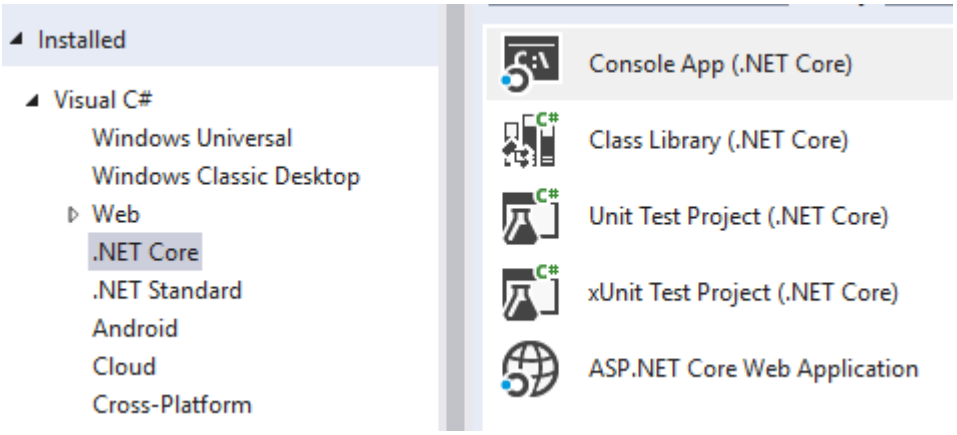
Core language features

Data types

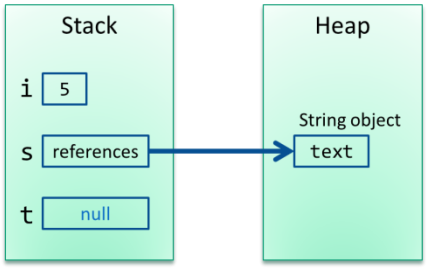


C# is a strongly-typed language. Before a value can be stored in a variable, the type of the variable must be specified.

Create a .NET Core Console app




```
int i = 5;
```



Value types store their contents in an area of memory called the stack. When the variable goes out of scope, because the method in which it was defined has finished executing, the value is discarded from the stack. Reference types, such as strings, are allocated in an area of memory called the heap. When the variable referencing an object becomes out of scope, the object becomes eligible for garbage collection.

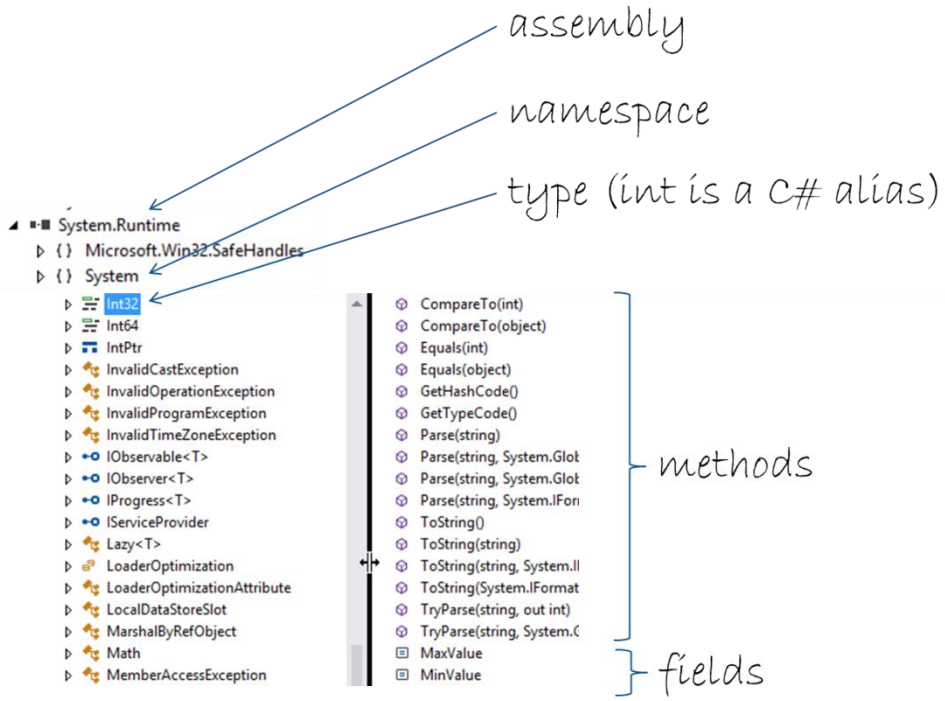
```
//[assembly: CLSCompliant(true)]
namespace Examples
{
    public class CLSCompliant
    {
        public uint i; //uint not in CLS

        public int I; //identifier differing only by case
    }
```

	Code	Description
⚠	CS3003	Type of 'CLSCompliant.i' is not CLS-compliant
⚠	CS3005	Identifier 'CLSCompliant.I' differing only in case is not CLS-compliant

https://docs.microsoft.com/en-us/dotnet/standard/language-independence-and-language-independent-components#Rules

public methods and properties that conform to the Common Language Specification can be accessed from code in assemblies written in any programming language that supports the CLS.

Assemblies and namespaces



String Class

Namespace: System

Assemblies: System.Runtime.dll, mscorlib.dll, netstandard.dll

List<T> Class

Namespace: System.Collections.Generic

Assemblies: System.Collections.dll, mscorlib.dll, netstandard.dll

Class	Namespace	Assembly		
		Core	**.NET**	**Standard**
String	System	System.Runtime	mscorlib	netstandard
List	System.Collections.Generic	System.Collections	mscorlib	netstandard
SqlConnection	System.Data.SqlClient	System.Data.SqlClient*	System.Data	System.Data.SqlClient*

*nuget

Operators

Category	Operator	Associativity
Primary	++ - - . [] ?. typeof	Left
Unary	+ - ! ~ ++ - - () await	Right
Multiplicative	* / %	Left
Additive	+ -	Left
Shift	<< >>	Left
Relational	< <= > >=	Left
Equality	== !=	Left
Logical AND	&	Left
Logical XOR	^	Left
Logical OR	|	Left
Conditional AND	&&	Left
Conditional OR	||	Left
Null-coalescing	??	Right
Conditional	?:	Right
Assignment	= += -= *= =>	Right

```
//null-coalescing operator returns the right hand operand
//when the left operand is null
string s = null;
Console.WriteLine(s??"Hello World!");

//null-conditional operator tests for null
//before accessing method or property
Console.WriteLine(s.Length);
Console.WriteLine(s?.Length);
```

Arithmetic series

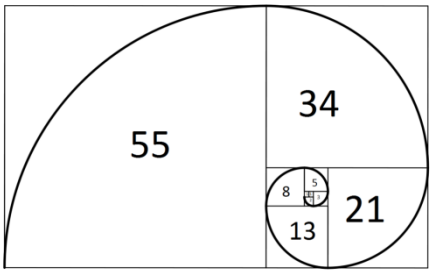
$$\sum_{k=0}^{10} \mathrm{k} = 0 + 1 + 2 \ldots$$

Geometric series

$$\sum_{k=0}^{10} 2^k = 2^0 + 2^1 + 2^2 \ldots$$

double d = Math.pow(2,1);

Fibonacci sequence



The sequence Fn of Fibonacci numbers is defined by

$F_n = F_{n-1} + F_{n-2}$ with seed values $F_1 = 1$, $F_2 = 1$

Using a loop, write the Fibonacci numbers below 100 to the console

F_n	F_{n-1}	F_{n-2}

```
for (int fn = 0, f1 = 1, f2 = 1; fn < 100; fn = f1 + f2)
{
      f2 = f1; //assign f n-1 to F n-2
      f1 = fn; //assign f n to = F n-1
      Console.WriteLine(fn);
}
```

Classes and objects

Static methods

Use the static modifier to declare a static member, which belongs to the type itself rather than to a specific object.

```
public class Maths
{
    public static double Factorial(int n)
    {
        double result = 1;
        for (; n > 1; n--)
        {
            result *= n;
        }
        return result;
    }
}
```

Recursion

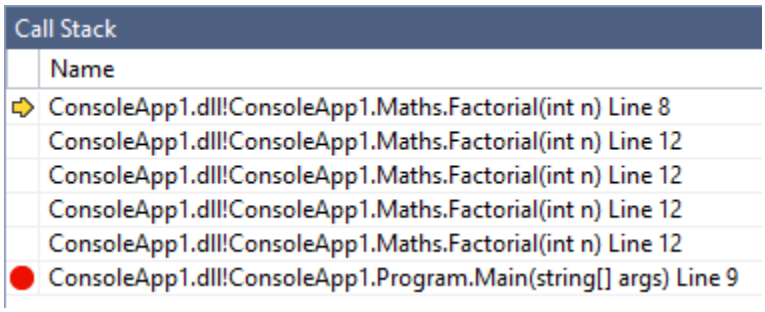
Two main components are required for every recursive function.

1. The base case returns a value without making any subsequent recursive calls. It does this for one or more special input values for which the function can be evaluated without recursion. For factorial(), the base case is n = 1.
2. The reduction step is the central part of a recursive function. It relates the value of the function at one (or more) input values to the value of the function at one (or more) other input values. Furthermore, the sequence of input values values must converge to the base case. For factorial(), the value of n decreases by 1 for each call, so the sequence of input values converges to the base case.

```
public class Maths
{
    public static double Factorial(int n)
    {
        if (n == 1){
            return 1; //base case
        }
        else
            return n * Factorial(n - 1);
    }
}
```

```
public class Maths
{
    public static double Factorial(int n)
    {
        return n==1 ? 1 : n * Factorial(n - 1);
    }
}
```

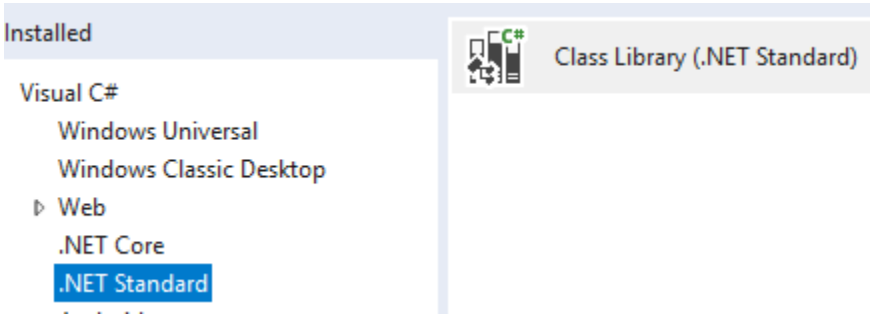
The method call stack can be viewed by opening Debug > Windows > Call Stack



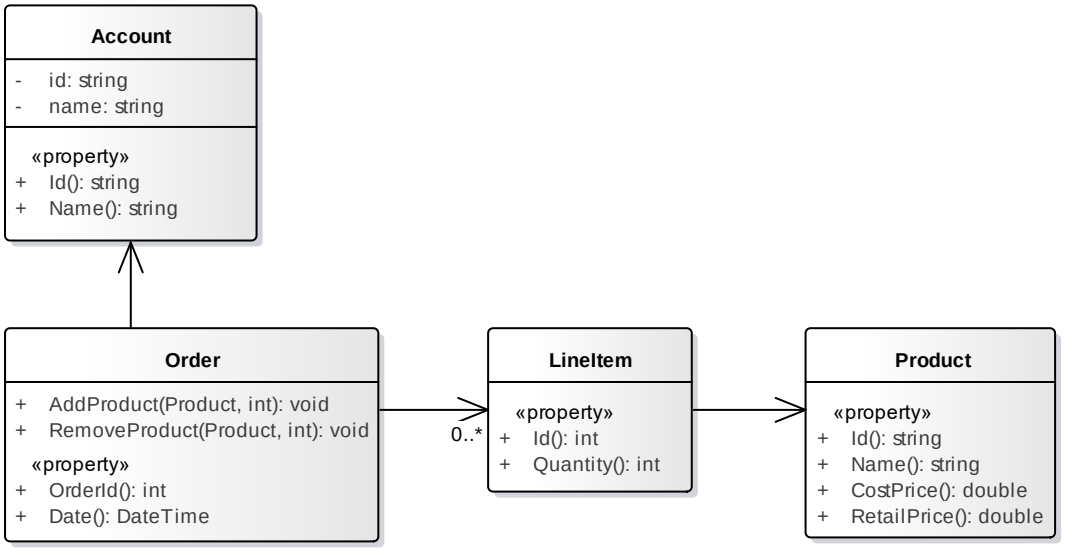
https://en.wikipedia.org/wiki/Roman_numerals

```
public static string ToRoman(int number)
{
    if (number == 1)
    {
        return string.Empty;//base case
    }
    if (number >= 1000) return "M" + ToRoman(number - 1000);
    if (number >= 900) return "CM" + ToRoman(number - 900);
    if (number >= 500) return "D" + ToRoman(number - 500);
    if (number >= 400) return "CD" + ToRoman(number - 400);
    if (number >= 100) return "C" + ToRoman(number - 100);
    if (number >= 90) return "XC" + ToRoman(number - 90);
    if (number >= 50) return "L" + ToRoman(number - 50);
    if (number >= 40) return "XL" + ToRoman(number - 40);
    if (number >= 10) return "X" + ToRoman(number - 10);
    if (number >= 9) return "IX" + ToRoman(number - 9);
    if (number >= 5) return "V" + ToRoman(number - 5);
    if (number >= 4) return "IV" + ToRoman(number - 4);
    if (number >= 1) return "I" + ToRoman(number - 1);
    return string.Empty;
}
```

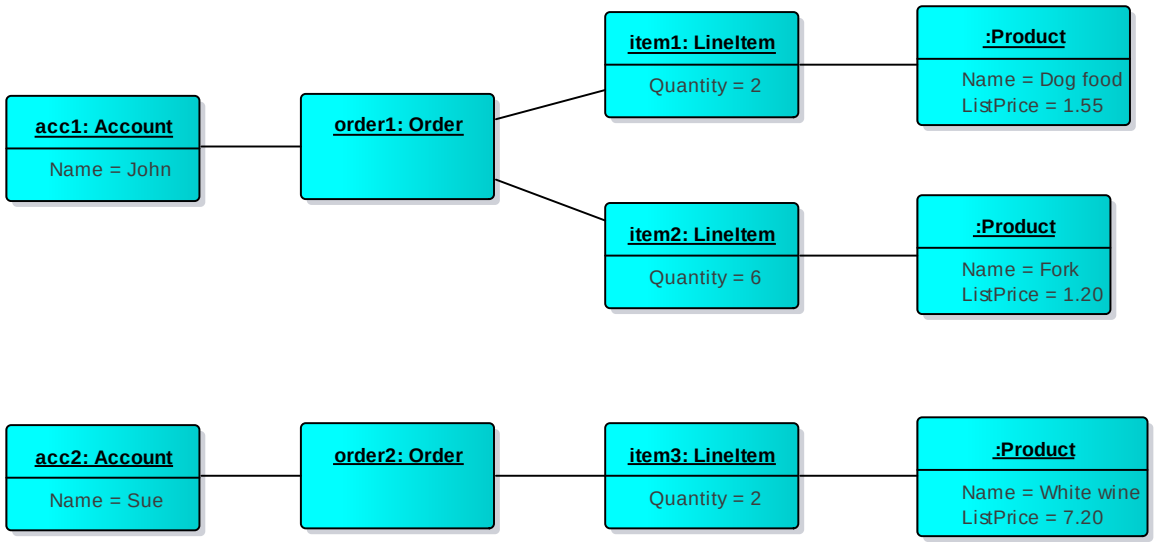
Add a .NET Standard project named ClassLibrary



UML class diagram

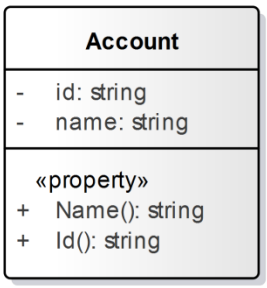


UML object diagram



Fields and properties

Account class



variables - state

methods and properties - behaviour

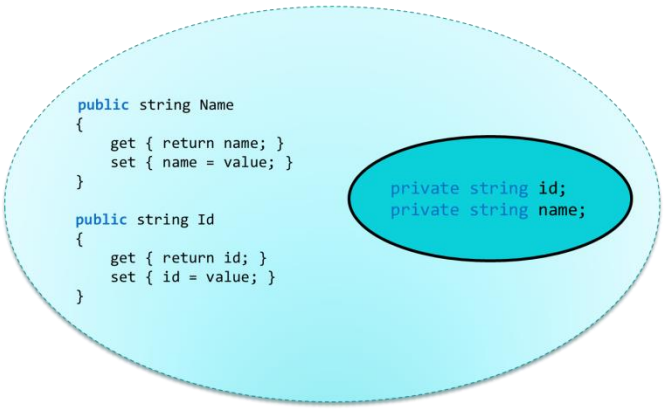
Use *propfull* snippet to generate field and property

```
public class Account
{
    private string id;
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public string Id
    {
        get { return id; }
        set { id = value; }
    }
}
```

state

behaviour

edit > refactor > encapsulate field (ctrl R ctrl E)



Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state.

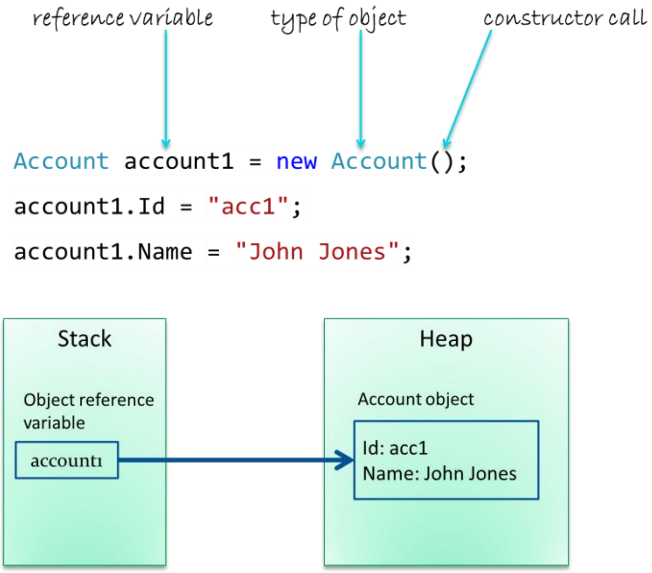
Accessibility	
public	Any referencing assembly
internal (the default)	Same assembly
private	Same class

Expression-bodied members

An expression body definition can be used where a method or property consists of a single expression.

```
public string Id { get => id; set => id = value; }
public string Name { get => name; set => name = value; }
```

Instantiating a class



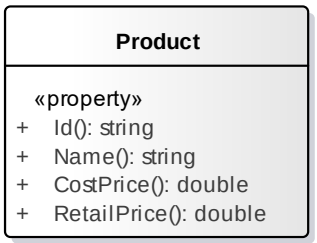
Build an instance of a class

- Add a reference to the class library project from the console application project
- Build an Account object and set its properties

```
using ClassLibrary;
namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Account account1 = new Account();
            account1.Id = "acc1";
            account1.Name = "John Jones";
            Console.WriteLine(account1.Id + " " + account1.Name);
```

Auto-implemented properties

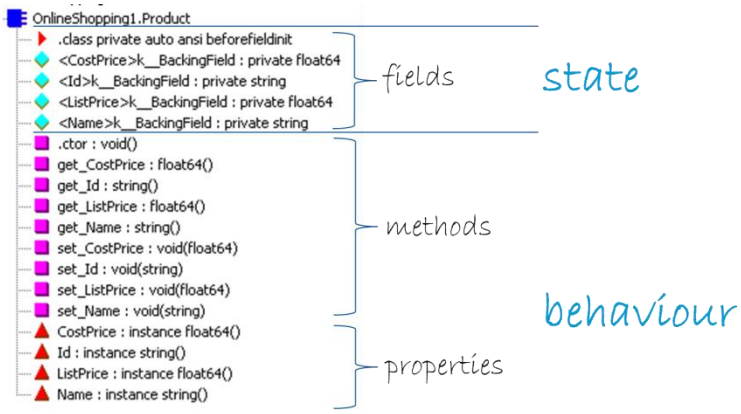
Product class



These property declarations are more concise and are appropriate when no additional logic required. The compiler creates a private, anonymous backing field.

```
namespace OnlineShopping
{
    public class Product
    {
        public string Id { get; set; }

        public string Name { get; set; }

        public double CostPrice { get; set; }

        public double RetailPrice { get; set; }
    }
}
```

prop Tab Tab



The backing field can be viewed by opening the compiled assembly with the intermediate language disassembler (ildasm)

Constructors

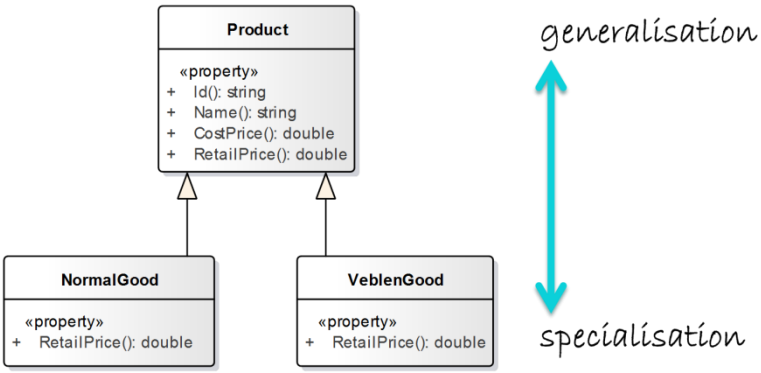
```
public class Product                    ctor Tab Tab
{
    public Product()
    {
    }
    public string Id { get; set; }
    public string Name { get; set; }
    public double CostPrice { get; set; }
    public double RetailPrice { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Product product1 = new Product();
```

Constructors are methods that are called when a class is instantiated. The empty brackets following new Product() indicates a call to the no-argument constructor. If there are no constructors in the source code, the compiler will generate a no-argument constructor.

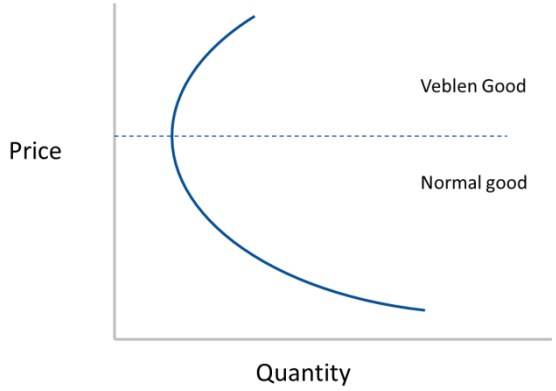
```
public class Product
{
    public Product()
    {
    }
    public Product(string id, string name,
                   double costPrice, double listPrice)
    {
        Id = id;
        Name = name;
        CostPrice = costPrice;
        RetailPrice = listPrice;
    }

class Program
{
    static void Main(string[] args)
    {
        Product product1 = new Product();
        Product product2 = new Product ("p2", "Fork", 0.4, 1.2 );
```

Overloading is the term used to describe multiple methods with the same name but different numbers of types of argument in a class. The above class contains overloaded constructors.

Inheritance



Inheritance enables creating new classes that reuse, extend, and modify the behaviour that is defined in other classes. A derived class is a specialization of the base class.



Veblen goods are types of luxury goods which are in demand because of the high prices asked for them. A reduction in price would reduce their cachet.

Overriding

```
public class Product
{
    public string Id { get; set; }

    public string Name { get; set; }

    public double CostPrice { get; set; }

    public virtual double RetailPrice { get; set; }
}
```

When a base class declares a method as virtual, a derived class can override the method with its own implementation.

The RetailPrice property is overridden in the derived classes of Product. The base class follows the colon in the following code.

```
public class NormalGood : Product
{
    public override double RetailPrice
    {
        get
        {
            return CostPrice * 2.5;
        }
    }
}
```

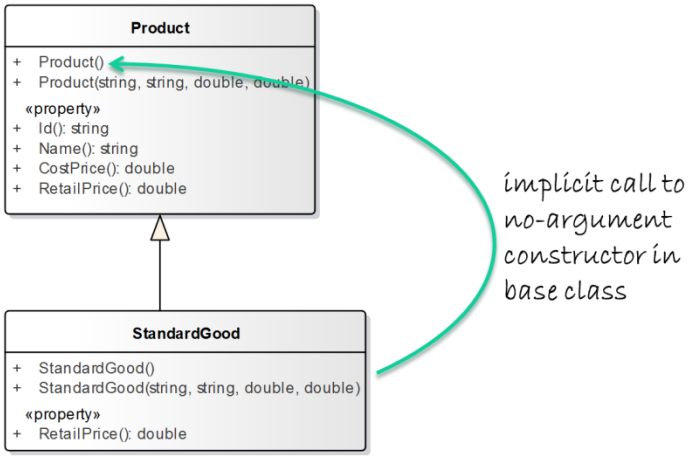
```
public class VeblenGood : Product
{
    public override double RetailPrice
    {
        get
        {
            return CostPrice * 5;
        }
    }
}
```

Accessibility	
public	Any referencing assembly
protected	Same class or derived class
internal (the default)	Same assembly
protected internal	Same assembly or derived class
private	Same class

Constructors in a hierarchy

```
public class NormalGood : Product
{
    public NormalGood()
    {

    }
    public NormalGood(string id, string name,
                      double costPrice, double listPrice)
    {

    }
```

Constructors aren't inherited, so the following derived class includes overloaded constructors.



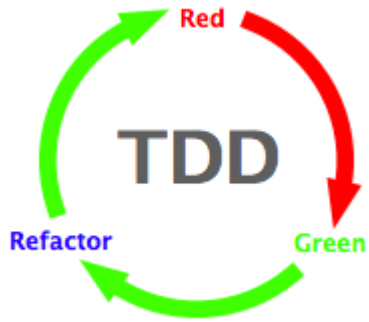
When a derived class is instantiated, the base class is also instantiated. By default, the no-argument constructor in the base class will be called.

The base keyword is used to invoke a specific constructor

```
public class NormalGood : Product
{
    public NormalGood()
    {

    }
    public NormalGood(string id, string name, double costPrice)
                            : base (id, name, costPrice, 0)
    {

    }
```

Unit Testing

Test-driven development



TDD is a software development process that relies on the repetition of a very short development cycle

1. Write an (initially failing) automated test case that defines a desired improvement or new function
2. Produce the minimum amount of code to pass that test
3. Refactor the new code to acceptable standards

Benefits

- Test cases force the developer to consider how functionality is used by clients, focussing on the interface before the implementation
- Helps to catch defects early in the development cycle
- Requires developers to think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces.
- Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Refactoring

- KIS (keep it simple) writing the smallest amount of code to make the test pass leads to simple solutions.
- YAGNI "You aren't going to need it" - avoid unnecessary methods.
- DRY (don't repeat yourself)
- SRP (single responsibility principle)

Types of test

Unit tests

- Single classes
- ensure high quality code
- replace real collaborators with mock objects

Integration tests

- the code under test is not isolated
- run more slowly than unit tests
- verify that modules are cooperating effectively
- Integration testing is similar to unit testing in that tests invoke methods of application classes in a unit testing framework. However, integration tests do not use mock objects to substitute implementations for service dependencies. Instead, integration tests rely on the application's services and components. The goal of integration tests is to exercise the functionality of the application in its normal run-time environment.

Acceptance tests

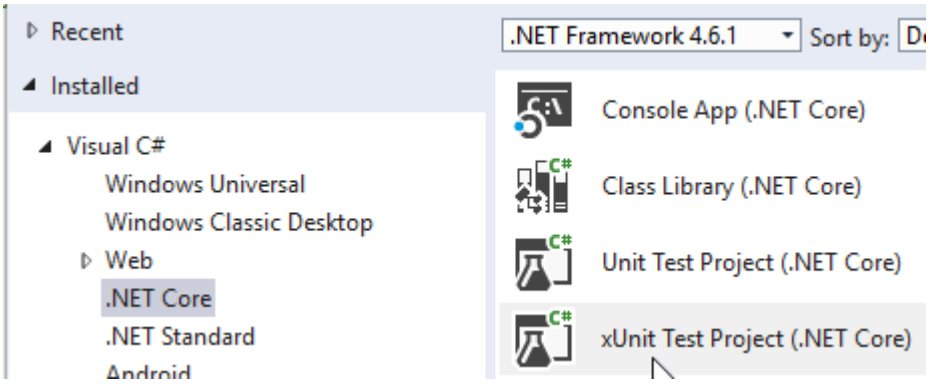
- multiple steps that represent realistic usage scenarios of the application as a whole.
- scope includes usability, functional correctness, and performance.

Phases when writing a test

1. Arrange – create objects
2. Act – execute methods to be tested
3. Assert – verify results

XUnit

Add a .NET Core xUnit Test project named ClassLibrary.Test



xUnit.net is an open source unit testing tool for the .NET framework, written by the original author of NUnit. See https://xunit.github.io/

A widely used naming convention for unit tests is to concatenate
[the name of the tested method]_[expected input / tested state]_[expected behaviour].

```
public class MathsTest
{
    [Fact]
    public void Factorial_WhenPassed5_ShouldReturn120 ()
    {
        //arrange
        //act
        //assert
    }
```

Facts and theories

Facts are tests which are always true. They test invariant conditions. Theories are tests which are only true for a particular set of data.

Parameters

```
namespace ClassLibrary1.Test
{
    public class MathsTest
    {
        [Fact]
        public void Factorial_WhenPassed5_ShouldReturn120()
        {
            //act
            double actual = Maths.Factorial(5);
            //assert
            Assert.Equal(120, actual);
        }

        [Theory]
        [InlineData(0, 1)]
        [InlineData(1, 1)]
        [InlineData(2, 2)]
        [InlineData(3, 6)]
        [InlineData(6, 720)]
        public void Factorial_ParameterizedTest(int n, double expected)
        {
            //act
            double actual = Maths.Factorial(n);
            //assert
            Assert.Equal(expected, actual);
        }

        [Theory]
        [InlineData(1, "I")]
        [InlineData(2, "II")]
        [InlineData(3, "III")]
        [InlineData(4, "IV")]
        [InlineData(58, "LVIII")]
        [InlineData(127, "CXXVII")]
        [InlineData(429, "CDXXIX")]
        public void ToRoman_ParameterizedTest(int n, string expected)
        {
            //act
            string actual = Maths.ToRoman(n);
            //assert
            Assert.Equal(expected, actual);
        }
```

Constructor

```
namespace ClassLibrary.Test
{
    public class ProductTest
    {
        [Fact]
        public void Constructor_WhenPassedParameters_ShouldSetProperties()
        {
            Product product = new Product("p1", "Dog's Dinner", 1.20, 2.50);
            Assert.Equal("p1", product.Id);
            Assert.Equal("Dog's Dinner", product.Name);
            Assert.Equal(1.20, product.CostPrice);
            Assert.Equal(2.50, product.RetailPrice);
        }
```

Arrays and Collections

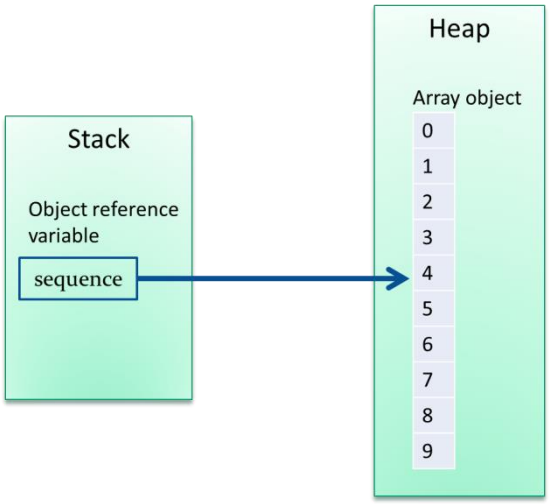
Arrays

Fibonacci

Arrays are objects; the expression

```
int[] sequence = new int[10];
sequence[9] = 34;
```

Builds an array of type int containing 10 elements, starting at element 0. The elements in an int array are initialised as 0; the elements in an array of a reference type, such as a string, are initialised as null (a null reference).



Add a method to the Maths class that takes an int argument and returns an array containing the first n Fibonacci numbers, where n is the argument to the method

```
namespace Examples.Test
{
    public class MathsTest
    {
        [Fact]
        public void Fibonacci_NumberOfElements_ShouldReturnCorrectSequence()
        {
            int[] expected = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
            int[] actual = Maths.Fibonacci(10);
            Assert.Equal(expected, actual);
        }
```

Object reference conversion

An object can be assigned to a base class variable, known as object reference conversion. The following expression uses object initialiser syntax to set the properties of the NormalGood object.

```
Product product1 = new NormalGood { Id = "p1", Name = "Dog Dinner", CostPrice = 0.4 };
```

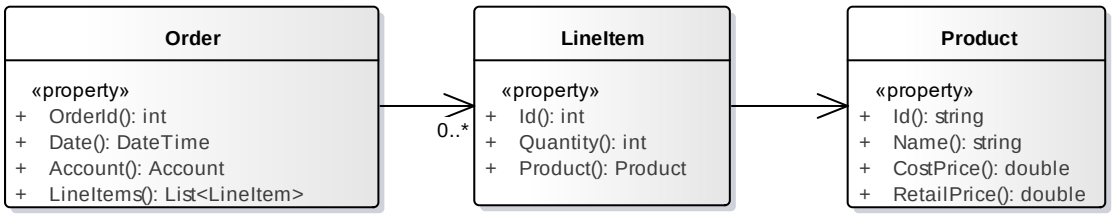
Polymorphism

Polymorphism can be demonstrated by calling the RetailPrice property of Product objects. Rather than calling the method that is defined by the variable's type, the overriding method in the NormalGood or VeblenGood class is invoked.

```
namespace ConsoleApplication.Examples
{
    public class Polymorphism
    {
        public static void Main()
        {
              Product product1 = new NormalGood { Id = "p1",
                                  Name = "Dog Dinner", CostPrice = 0.4 };
              Product product2 = new NormalGood { Id = "p2",
                                  Name = "Fork", CostPrice = 0.4 };
              Product product3 = new VeblenGood { Id = "p3",
                                  Name = "Krug Champagne", CostPrice = 25 };
              Product product4 = new VeblenGood { Id = "p4",
                                  Name = "Rolex watch", CostPrice = 700 };
              Product[] products = new Product[4];
              products[0] = product1;
              products[1] = product2;
              products[2] = product3;
              products[3] = product4;
              foreach (Product product in products)
              {
                  Console.WriteLine(product.RetailPrice);
              }
```

Collections

Order and LineItem classes



A LineItem has an associated Product and an Order is associated with multiple LineItems. The LineItems property is of type List<LineItem>. A List is a collection whose elements are accessible by a zero based index, like an array. The <LineItem> syntax declares the type of object that can be stored in the List.

```
public class Order
{
    public int OrderId { get; set; }
    public DateTime Date { get; set; } = DateTime.Now;
    public Account Account { get; set; }
    public List<LineItem> LineItems { get; set; } = new List<LineItem>();
}
```

Declaring the properties of a class type won't instantiate them. Either build the objects in the constructor or use a collection initializer.

Enum

The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list.

Add a property of type OrderStatus to the Order class

```
    public OrderStatus OrderStatus { get; set; } = OrderStatus.NotPlaced;
```

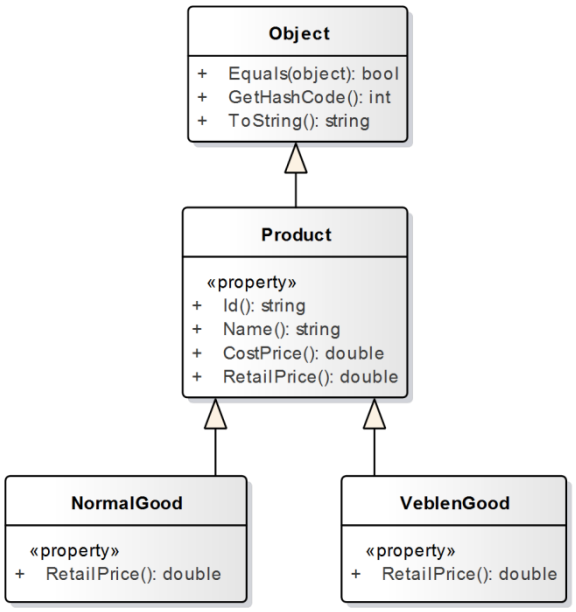
And then generate the following enum

```
public enum OrderStatus
{
    NotPlaced,
    New,
    Packed,
    Dispatched,
    Delivered
}
```

By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. An enum constant can be cast to an integral type.

```
int x = (int)OrderStatus.Dispatched;
```

Overriding Equals



The object class is at the top of the class hierarchy.

Classes can override the Equals method to enable object equality to be determined by value rather than by reference. For example, Product objects could be deemed to be equal if they have the same Id.

Classes that override the Equals method should also override the GetHashCode method. A hash code is a numeric value that is used to insert and identify an object in a hash-based collection. Equal objects should have equal hash codes, but unequal objects are not required to have unequal hash codes.

Start with some unit tests describing the expected behaviour

```
namespace ClassLibrary.Test
{
    public class ProductTest
    {
        [Fact]
        public void ProductsWithSameIdShouldBeEqual()
        {
            //arrange
            Product product1 = new Product { Id = "1" };
            Product product2 = new Product { Id = "1" };
            //act
            bool equal = product1.Equals(product2);
            //assert
            Assert.True(equal);
        }
```

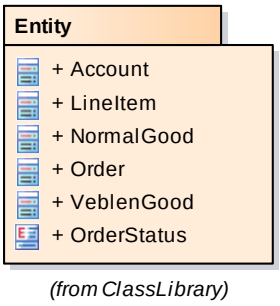
And then add override the Equals method in the Product class

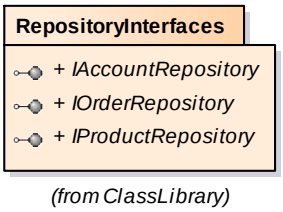
```
public class Product
{
    public override bool Equals(object obj)
    {
        return obj is Product && (obj as Product).Id == Id;
    }
```

Written as expression-bodied method

```
public override bool Equals(object obj) =>
            obj is Product && (obj as Product).Id == Id;
```

Interfaces





Entity classes and Repository Interfaces are shared by different implementations of the Data Layer.

Interfaces form a contract between the class and its users. An interface contains only the signatures of methods and properties.

```
namespace System.Data
{
    public interface IDbConnection
    {
        void Open();
        void Close();
    }
}
```

A class that implements an interface must include all its methods.

```
namespace System.Data.SqlClient
{
    public class SqlConnection
    {
        public void Open()
        {
        }
        public void Close()
        {
        }
    }
}
```

Generics

Generics introduce the concept of type parameters, which enable classes and methods to defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations.

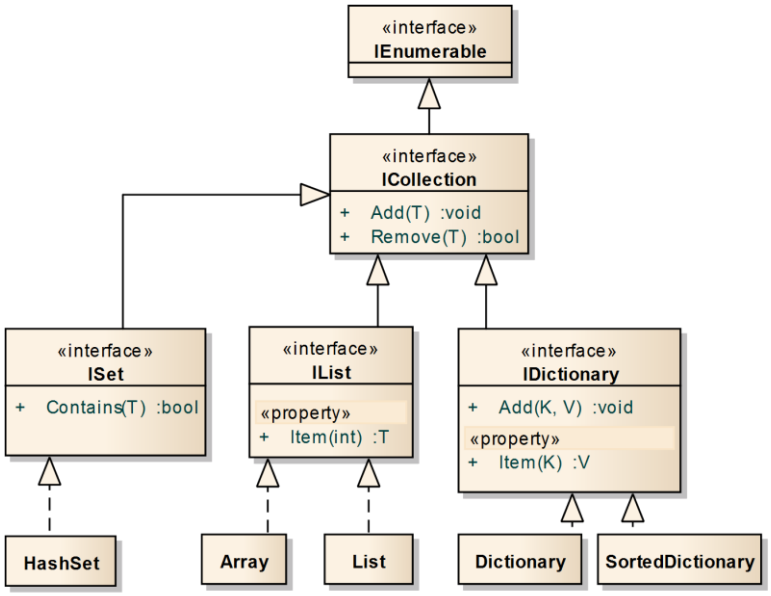
For example, the ICollection interface includes Add and Remove methods.

```
public interface ICollection<T>
{
    void Add(T item);
    bool Remove(T item);
}
```

A class that implements the interface must implement these methods

```
class HashSet<T> : ConsoleApplication.Examples.ICollection<T>
{
    public void Add(T item)
    {
        throw new NotImplementedException();
    }

    public bool Remove(T item)
    {
        throw new NotImplementedException();
    }
}
```

Collections framework

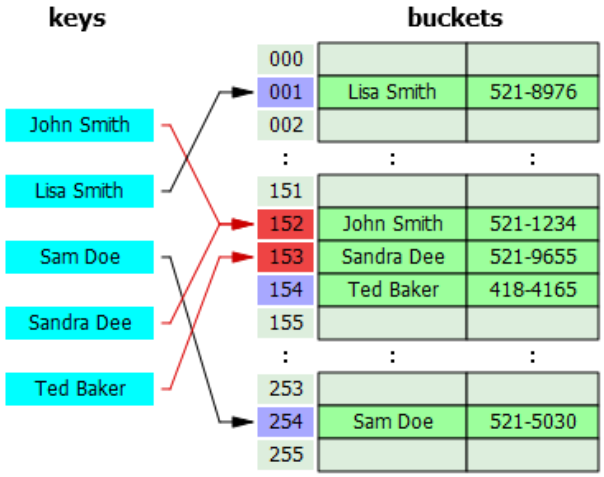


IEnumerable objects can be iterated through with a foreach loop. An ICollection is a very general collection. An ISet contains unique elements in no particular order. HashSet is an implementation of the ISet interface.

```
namespace ClassLibrary.Test
{
    public class ProductTest
    {
        [Fact]
        public void ProductsShouldWorkCorrectlyWithList()
        {
            List<Product> products = new List<Product>();
            Product product1 = new Product { Id = "1" };
            Product product2 = new Product { Id = "1" };
            products.Add(product1);
            Assert.True(products.Contains(product2));
        }

        [Fact]
        public void ProductsShouldWorkCorrectlyWithHashSet()
        {
            HashSet<Product> products = new HashSet<Product>();
            Product product1 = new Product { Id = "1" };
            Product product2 = new Product { Id = "1" };
            products.Add(product1);
            Assert.True(products.Contains(product2));
        }
    }
}
```

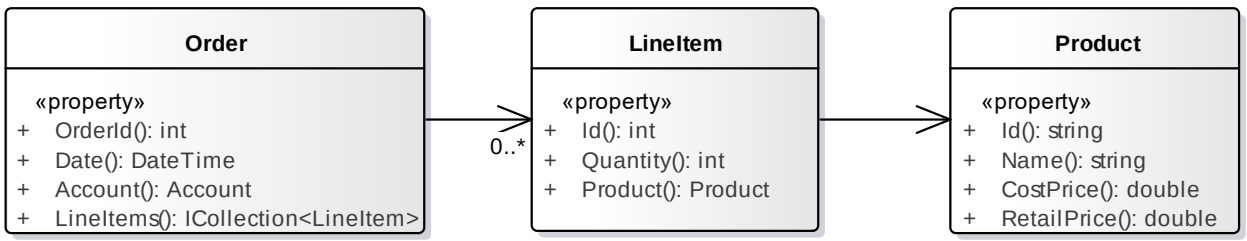
GetHashCode

A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. Instead, most hash table designs assume that hash collisions—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way. C# uses a strategy called open addressing, in which all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.



Property with Interface type

Changing the LineItems property from List to ICollection enables more flexibility in the implementation of the Order class.




```
public class Order
{
    public int OrderId { get; set; }
    public DateTime Date { get; set; } = DateTime.Now;
    public Account Account { get; set; }
    public ICollection<LineItem> LineItems { get; set; } = new List<LineItem>();
}
```

LINQ

Query syntax

Language-Integrated Query (LINQ) enables data from a variety of sources, such as a collection or a database, to be queried using a standard syntax. All LINQ query operations consist of three distinct actions:

1. Obtain the data source. This must be an object which implements the IEnumerable<T> interface
2. Create the query. This specifies what information to retrieve from the data source. Optionally, a query also specifies how that information should be sorted, grouped, and shaped before it is returned. The query expression contains three clauses: from, where and select.
    a. The from clause specifies the data source
    b. The where clause applies the filter,
    c. The select clause specifies the type of the returned elements.
3. Execute the query. The query variable itself only stores the query commands. The actual execution of the query is deferred until iterating over the query variable in a foreach statement.

- Data source

```
int[] numbers =  { 0, 1, 2, 3, 4, 5, 6 };
```

- Query creation

```
IEnumerable<int> numQuery =
       from num in numbers
       where (num % 2) == 0
       select num;
```

- Query execution

```
foreach (int num in numQuery)
{
    Console.WriteLine(num);
}
```

Similarly, to interrogate a collection of objects

- Data source

```
ICollection<Product> products = new List<Product> {
     new Product("p1", "Pedigree Chum", 0.70, 1.42),
     new Product("p2", "Knife", 0.60, 1.31),
     new Product("p3", "Fork", 0.75, 1.57),
     new Product("p4", "Spaghetti", 0.90, 1.92),
     new Product("p5", "Cheddar Cheese", 0.65, 1.47),
     new Product("p6", "Bean bag", 15.20, 32.20),
     new Product("p7", "Bookcase", 22.30, 46.32),
     new Product("p8", "Table", 55.20, 134.80),
     new Product("p9", "Chair", 43.70, 110.20),
     new Product("p10", "Doormat", 3.20, 7.40)
};
```

- Query creation

```
IEnumerable<string> result =
         from p in products
         where p.RetailPrice < 50
         select p.Name;
```

- Query execution

```
foreach (string s in result)
{
    Console.WriteLine(s);
}
```

Extension methods

Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code, there is no apparent difference between calling an extension method and the methods that are actually defined in a type. For example, the following class defines an extension method to the int type named IsPrime.

```
namespace PrimeNumbers
{
    public static class ExtensionMethod
    {
        public static bool IsPrime(this int i)
        {
            for (int j = 2; j < i; j++)
            {
                if (i % j == 0)
                    return false;
            }
            return true;
        }
    }
}

int i = 17;
Console.WriteLine(i.IsPrime());
```

There are extension methods of IEnumerable<T> in the System.Linq namespace that perform projections, filters and aggregate operations. Many of these methods take Delegate arguments.

Delegates

A delegate declaration defines a type that encapsulates a method with a particular set of arguments and return type. Delegates can be instantiated; the constructor takes a method argument. For example Delegate1 defines a delegate that can encapsulate a method that takes an int argument and returns an int.

```
namespace ConsoleApplication.Examples
{
    public delegate int Delegate1(int i);
}
```

To instantiate the delegate, pass a compatible method into the constructor. The method associated with the delegate can then be invoked.

```
    public class Delegates
    {
        public static int DoubleIt(int i)
        {
            return i * 2;
        }

        public static void Main()
        {
            Delegate1 d1 = new Delegate1(DoubleIt);
            Console.WriteLine(d1.Invoke(5));
            //or simply
            Console.WriteLine(d1(5));
        }
    }
```

Generic delegates enable the parameters and return type of the associated method to be determined by the developer. For example

```
namespace Examples
{
    public delegate TResult Delegate2<TSource, TResult>(TSource t);

    public class Delegates
    {
        public static bool IsEven(int i)
        {
            return i % 2 == 0;
        }

        public static void Main()
        {
            Delegate2<int, bool> d2 = new Delegate2<int, bool>(IsEven);
            Console.WriteLine(d2(7));
        }
    }
}
```

Lambda expressions

A lambda expression is an anonymous function and it is mostly used to create delegates in LINQ. Simply put, it's a method without a declaration, i.e., access modifier, return value declaration, and name. For example, the IsEven method could be written as a Lambda expression and passed into the delegate's constructor. The Lambda operator `=>` separates the parameter of the anonymous method from its content.

```
namespace ConsoleApplication.Examples
{
    public delegate TResult Delegate2<TSource, TResult>(TSource t);

    public class Delegates
    {
        public static void Main()
        {
            Delegate2<int, bool> d3 = new Delegate2<int, bool>(i => i % 2 ==0);
            Console.WriteLine(d3(7));
        }
    }
}
```

The System namespace defines a number of delegates, including Func

```
namespace ConsoleApplication.Examples
{
    public class Delegates
    {
        public static void Main()
        {
            Func<int, bool> func = new Func<int, bool>(i => i % 2 == 0);
            Console.WriteLine(func(7));
        }
    }
}
```

LINQ method syntax

```
 public static void MethodSyntax1()
 {
     ICollection<Product> products = new List<Product> {
         new Product("p1", "Pedigree Chum", 0.70, 1.42),
         new Product("p2", "Knife", 0.60, 1.31),
     };
     //The Where and Select extension methods of IEnumerable<T> both take Func
     //arguments
     IEnumerable<string> result = products.
              Where(p => p.RetailPrice < 50).
              Select(p => p.Name);
     result.ToList().ForEach(n => Console.WriteLine(n));

     int count = products.Count(p => p.RetailPrice > 50);

     //what is the average percentage markup on the cost price ?
     double percent = products.Average(
                               p => (p.RetailPrice / p.CostPrice) - 1) * 100;
     Console.WriteLine($"{percent:F1}%");
 }
```

An aggregation operation computes a single value from a collection of values. Methods of the IEnumerable interface include Average, Max, Min, Count, FirstOrDefault, SingleOrDefault

Method Group

Method groups provide a simple syntax to assign a method to a delegate variable. If a lambda expression consists of only one method, it can be written as a method group to achieve more compact syntax and prevent compile-time overhead caused by using lambdas

```
Action<string> action = Console.WriteLine; //overloaded group of methods

IEnumerable<string> result = products.Where(p => p.RetailPrice < 50)
                                     .Select(p => p.Name);

result.ToList().ForEach(action); //delegate instance

result.ToList().ForEach(Console.WriteLine); //method group
```

Anonymous types

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. You create anonymous types by using the new operator together with an object initializer. Typically, when you use an anonymous type to initialize a variable, you declare the variable as an implicitly typed local variable by using var.

```
var result = products.
    Where(p => p.RetailPrice < 50).
    Select(p => new { p.Id, p.Name } );

result.ToList().ForEach(x => Console.WriteLine(x.Name));
```

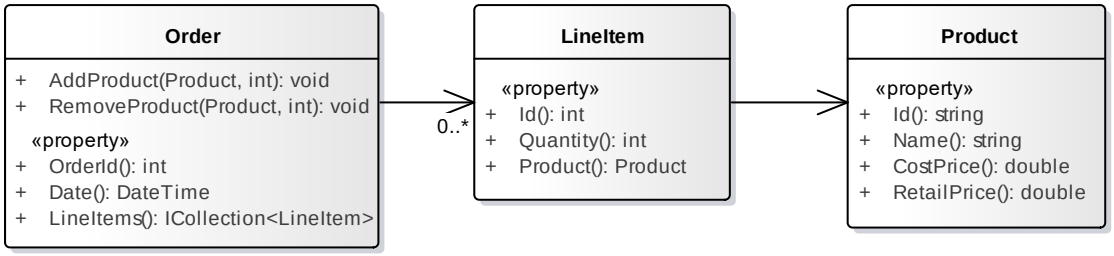
Adding Products to an Order

Specification

The AddProduct method should do the following:

1. If the LineItem contains a Product that is already in the Order, increment the LineItem's quantity
2. Otherwise, add a new LineItem to the ICollection of LineItems in the order

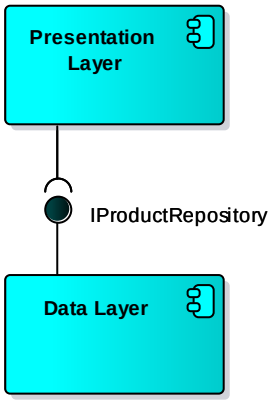
Test Driven Development




```
namespace ClassLibrary.Test
{
    public class OrderTest
    {
        [Fact]
        public void AddProduct_WhenCalledTwiceWithSameProduct_
                                                ShouldIncrementLineItemQuantity()
        {
            //arrange
            Order order = new Order();
            Product product1 = new Product("p1", "Dog Dinner", 1.20);
            Product product2 = new Product("p1", "Dog Dinner", 1.20);
            //act
            order.AddProduct(product1, 2);
            order.AddProduct(product2, 2);
            //assert
            Assert.Equal(1, order.LineItems.Count);
            Assert.Equal(4, order.LineItems.First().Quantity);
        }
```

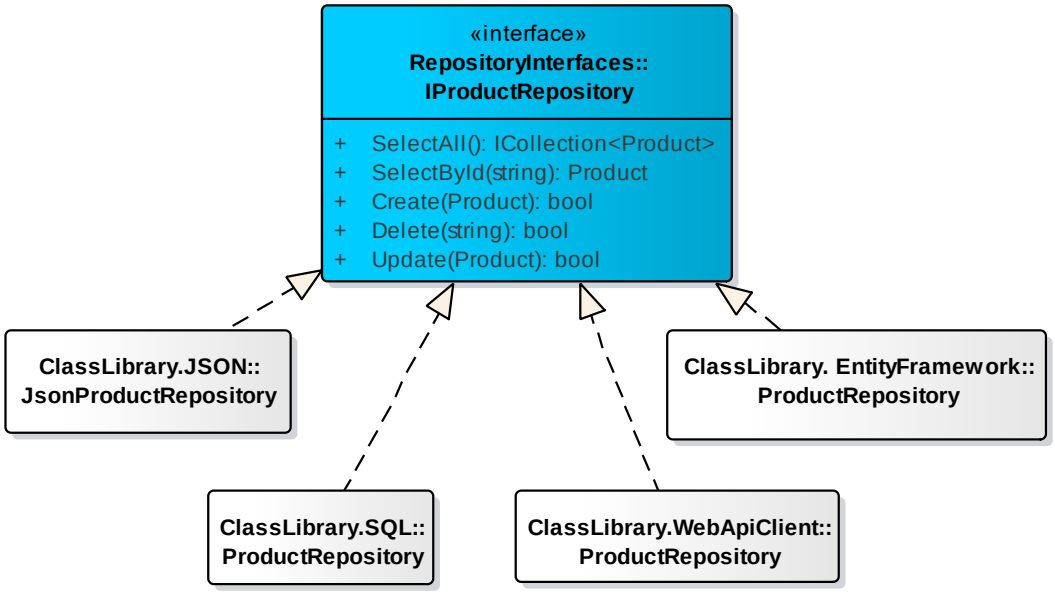
First, generate the AddProduct method and run the initially failing tests. Next, complete the methods so that the tests pass. Finally, refactor the code if required.

Decoupling layers



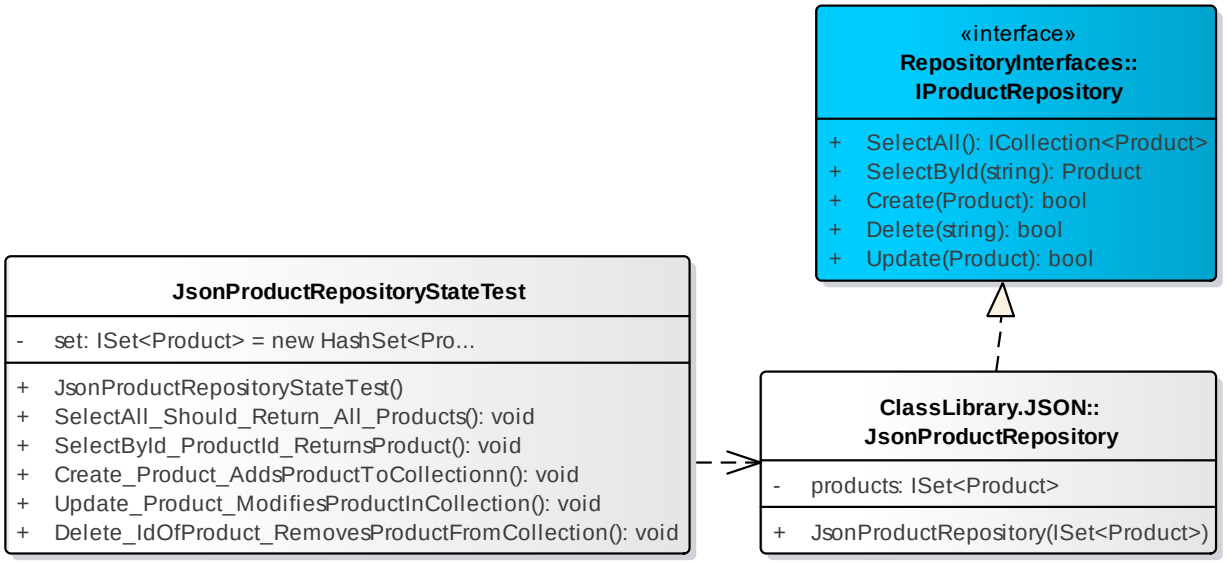
The UML component diagram shows two components connected by an interface, IProductModel. An interface is a contract between components, providing just a specification for properties and methods, deferring implementation details to a class.

The IProductRepository interface can have a number of implementations



JsonProductRepository

Add a .NET Standard class library project named ClassLibrary.JSON. The JsonProductRepository class stores products in a Set, which could be persisted by serializing the collection as JSON to a file.



Passing by reference

The out, ref and in keywords enable passing by reference. https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/method-parameters

```
public class RefExample
{
    public static void Main()
    {
        int number; //argument passed to out parameter must be modified by called
                                                                                method
        string str = "5";
        bool success = Int32.TryParse(str, out number);

        int val = 1; //argument passed to a ref parameter must be initialized
        PassByRef(ref val);
        Console.WriteLine(val);

        int constant = 5; //in arguments are constants that can't be modified by
                                                                    the called method
        PassConstByRef(constant);

        Console.ReadKey();
    }
    public static void PassByRef(ref int i)
    {
        i = i + 44;
    }

    //C# 7.2 (properties > build > advanced)
    public static void PassConstByRef(in int i)
    {
        //i = i + 44;
    }
}
```

Variance

```
public delegate TResult Func<in T, out TResult>( T arg )
```

contravariant
T can be less derived

covariant
TResult can be more derived

---

contravariant
T can be less derived

covariant
TResult can be more derived

```
namespace Examples
{
    class Variance
    {
        static void Main(string[] args)
        {
            Func<Product, Product> func =
                                new Func<Product, Product>(Target);
        }

        private static VeblenGood Target(Object arg)
        {
            throw new NotImplementedException();
        }
    }
}
```

File Handling and Exceptions

Exceptions

Unhandled Exceptions

The following method would terminate with an unhandled exception

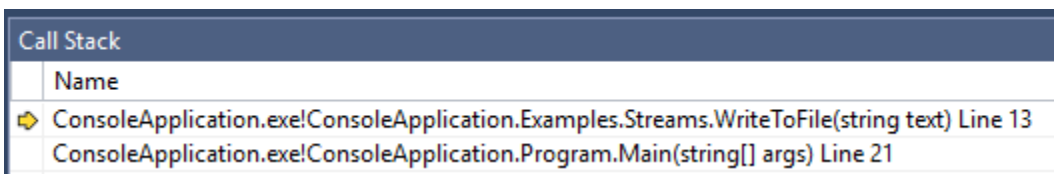
```
namespace Examples
{
    class Streams
    {
        public static void WriteToFile(string text)
        {
            //UnauthorizedAccessException
            File.WriteAllText(@"C:\file.txt", text);
            //DirectoryNotFoundException
            File.WriteAllText(@"Z:\file.txt", text);
        }
    }
```

Catching Exceptions

Exception handling is the process of responding to the occurrence of exceptional conditions requiring special processing, often changing the normal flow of program execution. The documentation for the Streams.WriteToFile method indicates that it can throw a number of Exceptions, including UnauthorizedAccessException and DirectoryNotFoundException.

```
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Streams.WriteToFile("something");
            }
            catch (UnauthorizedAccessException e)
            {
                Console.WriteLine(e);
            }
            catch (DirectoryNotFoundException e)
            {
                Console.WriteLine(e);
            }
```

Debug > Windows > Call Stack



Exceptions are thrown down the call stack until they're caught

Throwing Exceptions

```
namespace Examples.Test
{
    public class MathsTest
    {
        [Fact]
        [Trait("Category", "Unit Test - State")]
        public void Factorial_NegativeNumber_ShouldThrowArgumentOutOfRangeException ()
        {
            Assert.Throws<ArgumentOutOfRangeException>(() => Maths.Factorial(-1));
        }
    }
}
```

Assert.Throws takes a The Func<object> delegate. This encapsulates a method (the test code) that has no parameters and returns an object.

```
namespace Examples
{
    public class Maths
    {
        public static double Factorial(int n)
        {
            if (n < 0)
                throw new ArgumentOutOfRangeException("argument can't be negative");
```

Move the following test from Green to Red

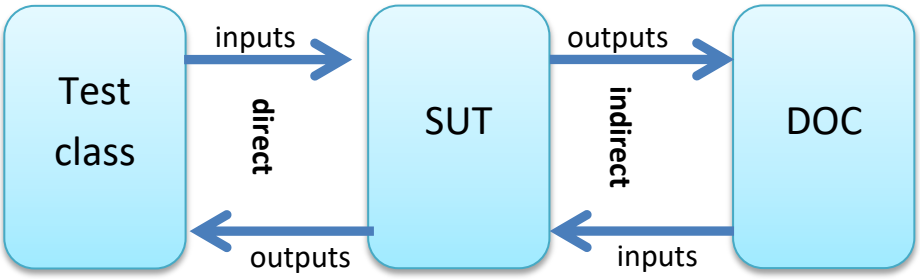
```
namespace ClassLibrary.Test
{
   public class OrderTest
   {
        [Fact]
        public void RemoveProduct_WhenPassedProductNotInLineItem_
                                                            ShouldThrowException()
        {
            //arrange
            Order order = new Order();
            order.LineItems = new List<LineItem> {
                new LineItem(new Product("p1", "Dog Dinner", 1.20), 5),
                new LineItem(new Product("p2", "Cutlery", 5.20), 2)
            };

            //act
            //assert
            Assert.Throws<InvalidOperationException>(()=>
                order.RemoveProduct(new Product("p3", "Cat Food", 1.15), 1)
            );
        }
```

Object serialization

Serialization is the process of converting the state of an object into a form that can be persisted or transported. The reverse is deserialization.

Binary serialization and XML serialization are part of the framework. A third party framework can be used for JSON serialization.

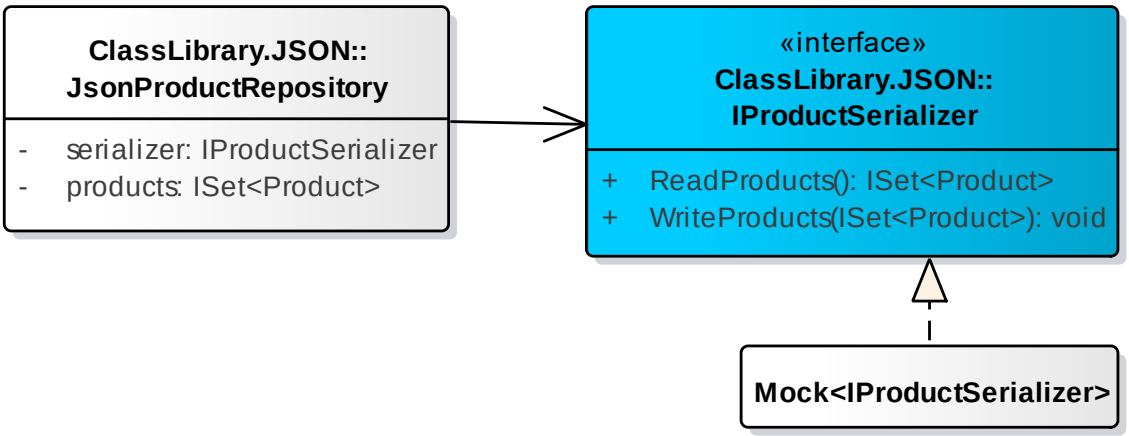
Interactions tests



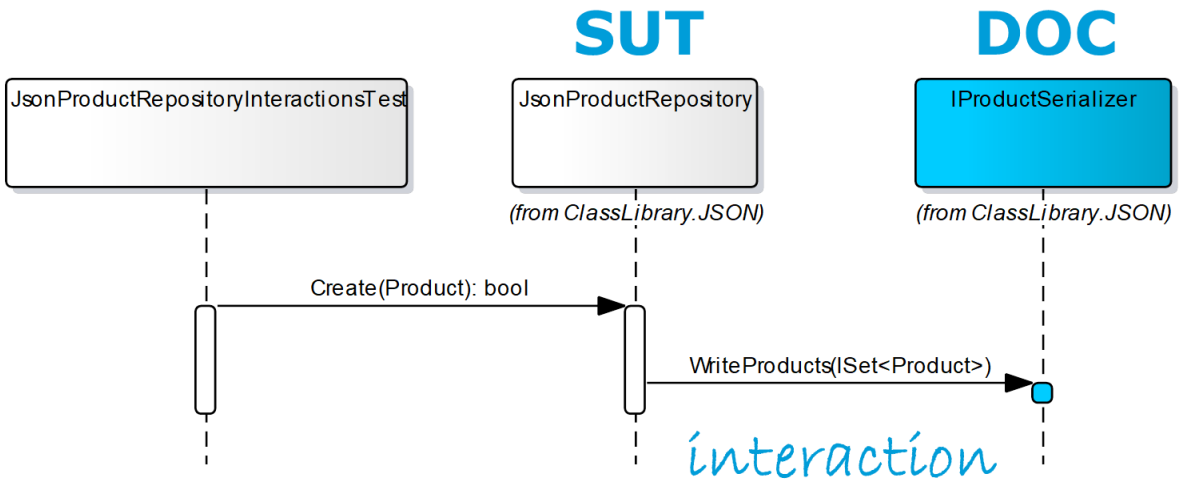
Unit tests apply to classes in isolation; the System Under Test (SUT) in the above diagram. They are intended to run fast and to pinpoint bugs with accuracy. **State testing** involves writing tests for direct inputs and outputs, while **interactions testing** verifies the way that the SUT interacts with collaborators (Depended On Components or DOCs).

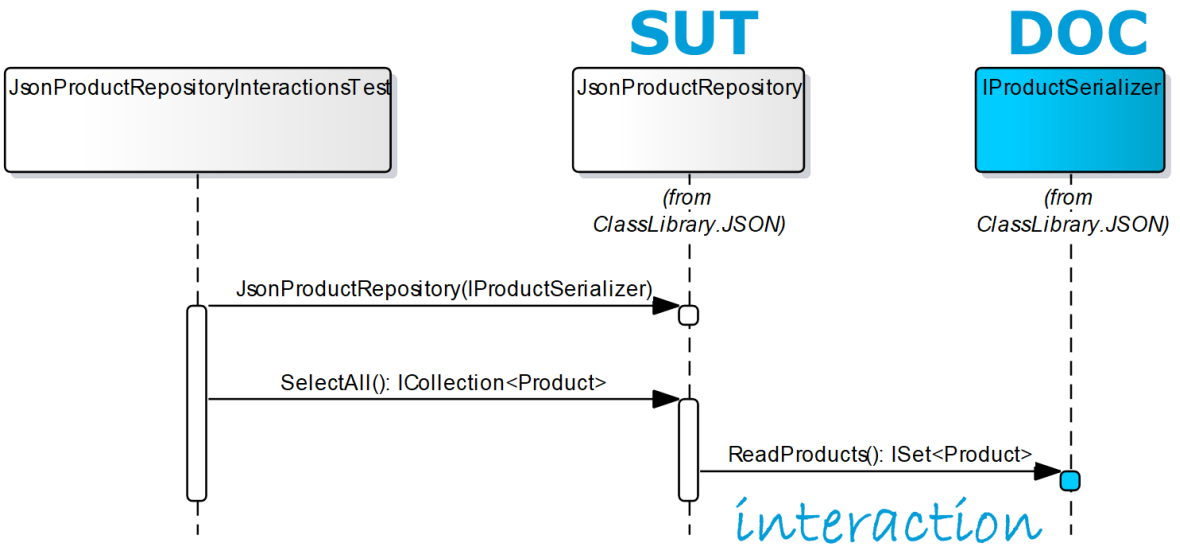
Test doubles look and behave like their release-intended counterparts, but are actually simplified versions. Mocks are a category of test double that's pre-programmed with expectations which form a specification of the calls they are expected to receive.

Moq is a library for creating mock objects and verifying interactions between them and the SUT. For example, a mock implementation of IProductSerializer could be used to verify the interactions with the CollectionProductModel.



The sequence diagrams show the expected interactions

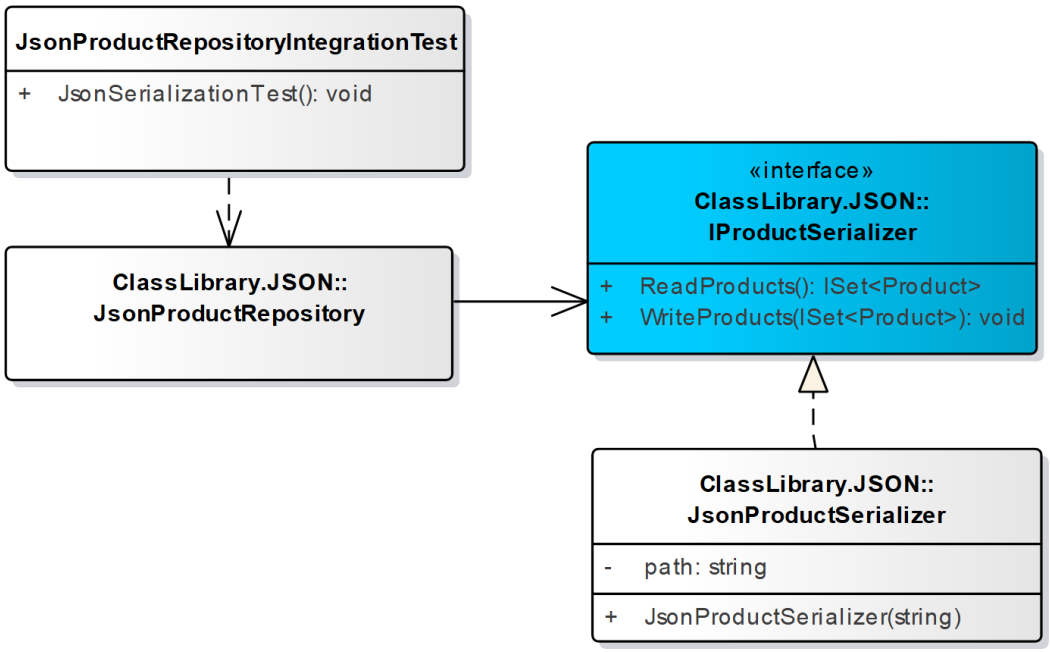




See `ClassLibrary.JSON.Test`

Integration tests

Unit tests focus on individual classes, while integration tests focus on the integration of different modules. This example tests the methods of the CollectionProductModel class, which uses the JsonProductSerializer class to persist the ICollection to a file.



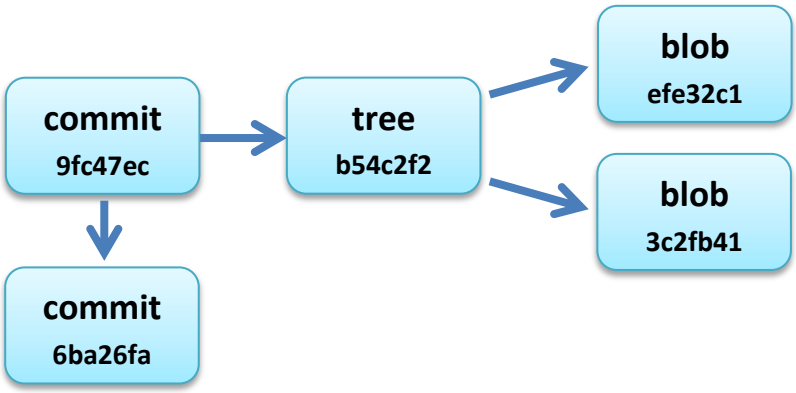

```
//Requires Newtonsoft.Json.dll from NuGet

namespace ClassLibrary.JSON
{
    public class JsonProductSerializer : IProductSerializer
    {
        private string path;
        public JsonProductSerializer(string path)
        {
            this.path = path;
        }

        public void WriteProducts(ISet<Product> products)
        {
            string output = JsonConvert.SerializeObject(products);
            File.WriteAllText(path, output);
        }

        public ISet<Product> ReadProducts()
        {
            if (File.Exists(path))
            {
                return JsonConvert.DeserializeObject<ISet<Product>>
                                                        (File.ReadAllText(path));
            }
            return new HashSet<Product>();
        }
    }
}
```

Workflow

Git is a distributed revision control system with an emphasis on speed, data integrity and support for distributed, non-linear workflows.

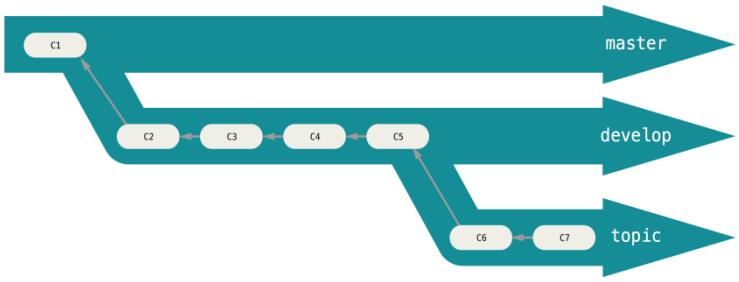
Every Git working directory is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server.

1. Create a local repository
    - `>git init`
2. Add a .gitignore file, listing files that Git should ignore
3. Add files to the index. SHA-1 checksum is generated for each file and the bytes are stored in the repository as a blob
    - `>git add *`
4. commit files in the index; this creates a snapshot of the project
    - `>git commit –m “message”`
    - tree object with checksum is generated for project directories
    - commit object with checksum includes metadata and points at root directory and previous commit



5. Optionally add a remote repository
    - `>git remote add origin https://dineen701@bitbucket.org/dineen701/.net-course.git`
6. Push to the remote
    - `>git push origin master`
7. Create and checkout develop branch
    - `>git branch develop`
    - `>git checkout develop`

A branch is a pointer to a commit. The default branch name in Git is **master**. A pointer called **HEAD** tracks the current branch. A workflow might maintain a master branch for stable code that will be released; a branch named develop to test stability and short-lived topic branches



Merging

Id	Message
f6adda2	**develop** HEAD C2
ea3363c	master Main method

1. Checkout master
   - `>git checkout master`
2. and merge in the develop branch
   - `>git merge develop`

Id	Message
f6adda2	develop **master** HEAD C2
ea3363c	Main method

This is called a fast forward merge, as there's no divergent work to merge together

3. Merge branches that have updated the same file
   - `>git merge develop`

This results in a conflict. The version in HEAD (the master branch) contains changes that conflict with the develop branch. Edit the file

```
<<<<<<< HEAD
        Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).info("Hello");
        //modified in master branch
=======
        Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).info("Hello");
        //added in develop branch
>>>>>>> refs/heads/develop
```

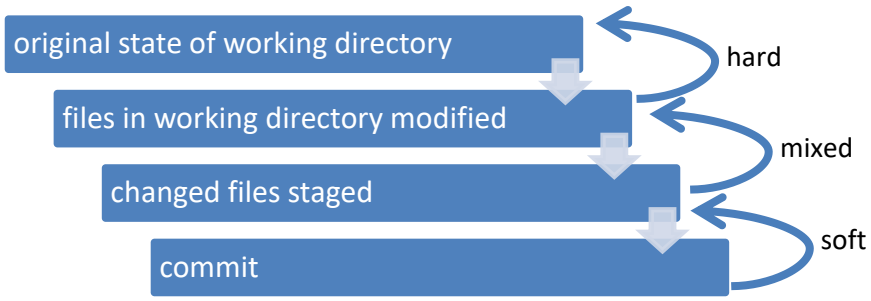
Save the amended file, then stage and commit

- `> git add *`
- `> git commit –m 'conflict resolved'`

Reset

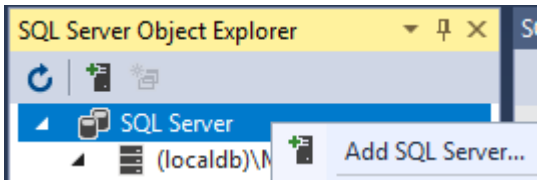
Checkout moves HEAD, the pointer to the current branch. This changes the working directory but does not delete previous commits

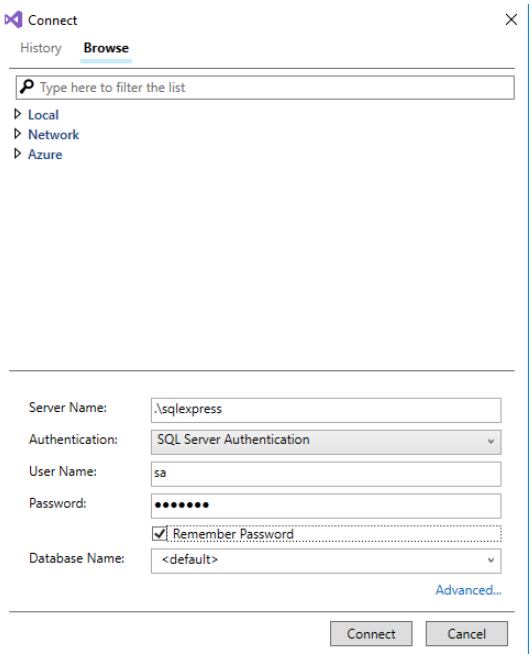
1. A **soft reset** undoes previous commits, moving the files to the index (see git staging area). The working directory is unchanged.
   a. `>git reset –soft ea3363c`
2. A **mixed reset** differs from a soft reset in that changes are unstaged
3. A **hard reset** undoes changes to the working directory. Unlike checking out a commit, this is not reversible.



SQL

Adding a database using Visual Studio





To create and populate a table named Accounts in a SqlServer database, run the following script (which is in the Create folder of the Database project)

```
use store;

create table Accounts (
	Id nvarchar(128) not null primary key,
	Name nvarchar (max)
);

insert into accounts (id, name) values ('acc1','John Smith');
insert into accounts (id, name) values ('acc2','Jane Jones');
insert into accounts (id, name) values ('acc3','Brian Johnson');
insert into accounts (id, name) values ('acc4','Sue Smedley');

select * from accounts;
```

Some Transact-SQL Data Types

Data Type Categories	SQL Data types	C# type
Character strings	varchar	string
Unicode character strings	nvarchar	string
Exact numerics	int	int
Approximate numerics	float	double
Date and time	datetime2	DateTime
Binary data	varbinary	byte[]
Other Data Types	rowversion	byte[]

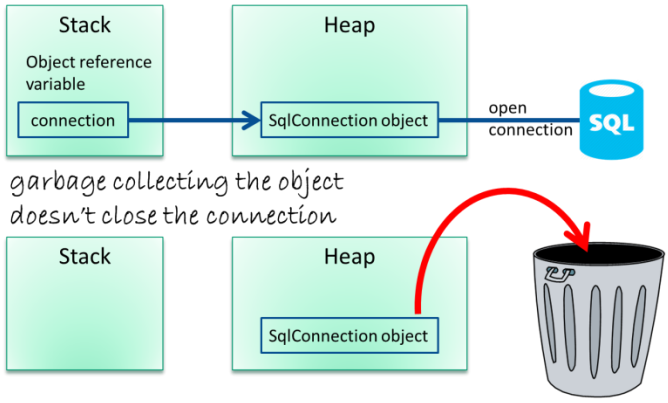
Connecting to SqlServer

The connection string can be stored a settings file. This stores key-value pairs. Open the project properties and select Setings.

```
Namespace Examples
{
    public class ConnectToDatabase
    {
        public static void Main()
        {
            string connectionString = "Data Source=.\\sqlexpress;
                                Initial Catalog=ecommerce;User ID=sa;Password=carpond";
            SqlConnection connection = new SqlConnection(connectionString);
            connection.Open();
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = connection;
            cmd.CommandText = "insert into accounts (id, name) values
                                                            ('acc1','John Smith');";
            int rowsInserted = cmd.ExecuteNonQuery();
            connection.Close();
        }
    }
}
```

Unmanaged Resources

Objects that wrap operating system resources, such as files, windows, network connections, or database connections are known as unmanaged resources, as they're not managed by the CLR. Objects that no longer have a reference become eligible for removal from the heap (garbage collection). Removing a SqlConnection object from memory won't close the connection to the database though.




```
string connectionString = new Settings().sqlserver;
SqlConnection connection = new SqlConnection(connectionString);
connection.Open();
SqlCommand cmd = new SqlCommand();
cmd.Connection = connection;
cmd.CommandText = "insert into account (id, name) values ('acc1'
int rowsInserted = cmd.ExecuteNonQuery();
connection.Close();
```

SqlException was unhandled

By placing the expression that can cause an exception in a try block, the following finally block which is always executed can be used to close the connection.

```
string connectionString = "Data Source=.\\sqlexpress;
                          Initial Catalog=ecommerce;User ID=sa;Password=carpond";
SqlConnection connection = new SqlConnection(connectionString);
try
{
    connection.Open();
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = connection;
    cmd.CommandText = "insert into account (id, name) values
                                                    ('acc1','John Smith');";
    int rowsInserted = cmd.ExecuteNonQuery();
}
finally
{
     connection.Close();
}
```

Using blocks

A more concise alternative structure is to include a using block. This can be used with an object that implements the IDisposable interface. When the block exits, the unmanaged resource will be closed.

```
public static void Main()
{
    using (//an IDisposable object)
    {
        //connection to unmanaged resource is closed
        //when using block exits
    }
}
```

Although the above code closes the unmanaged resource, it won't handle any exceptions, so a try and catch block can be included.

```
namespace ConsoleApplication.Examples
{
    public class ConnectToDatabase
    {
        public static void UsingBlock()
        {
            string connectionString = Settings.Default.sqlserver;
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                using (SqlCommand cmd = new SqlCommand())
                {
                    cmd.Connection = connection;
                    cmd.CommandText = "insert into account (id, name)
                                                    values ('acc1','John Smith');";
                    try
                    {
                        int rowsInserted = cmd.ExecuteNonQuery();
                    }
                    catch (SqlException e)
                    {
                        Console.WriteLine(e.Message);
                    }
                }
            }
```

Transactions

To ensure that multiple updates leave the database in a consistent state, a transaction can be started and either committed or rolled back. Transactions other than *Read Uncommitted* use row-level locking, and both will wait if they try to change a row updated by an uncommitted concurrent transaction (pessimistic concurrency)

```
public static void WithTransactions()
{
    string connectionString = Settings.Default.sqlserver;
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        using (SqlCommand cmd = new SqlCommand())
        {
            SqlTransaction transaction = connection.BeginTransaction(
                                                IsolationLevel.Serializable);
            cmd.Connection = connection;
            cmd.Transaction = transaction;
            try
            {
                //valid expression
                cmd.CommandText = "insert into accounts (id, name)
                                                values ('acc1','John Smith');";
                int rowsInserted = cmd.ExecuteNonQuery();
                //invalid expression
                cmd.CommandText = "update account set name = 'Jane Smith'
                                                where id = 'acc1';";
                int rowsUpdated = cmd.ExecuteNonQuery();
                transaction.Commit();
            }
            catch (SqlException e)
            {
                transaction.Rollback();
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Isolation Level	Read	Update	Insert
Read Uncommitted	Reads data which is yet not committed.	Allowed	Allowed
Read Committed (Default)	Reads data which is committed.	Allowed	Allowed
Repeatable Read	Reads data which is committed.	Not Allowed	Allowed
Serializable	Reads data which is committed.	Not Allowed	Not Allowed

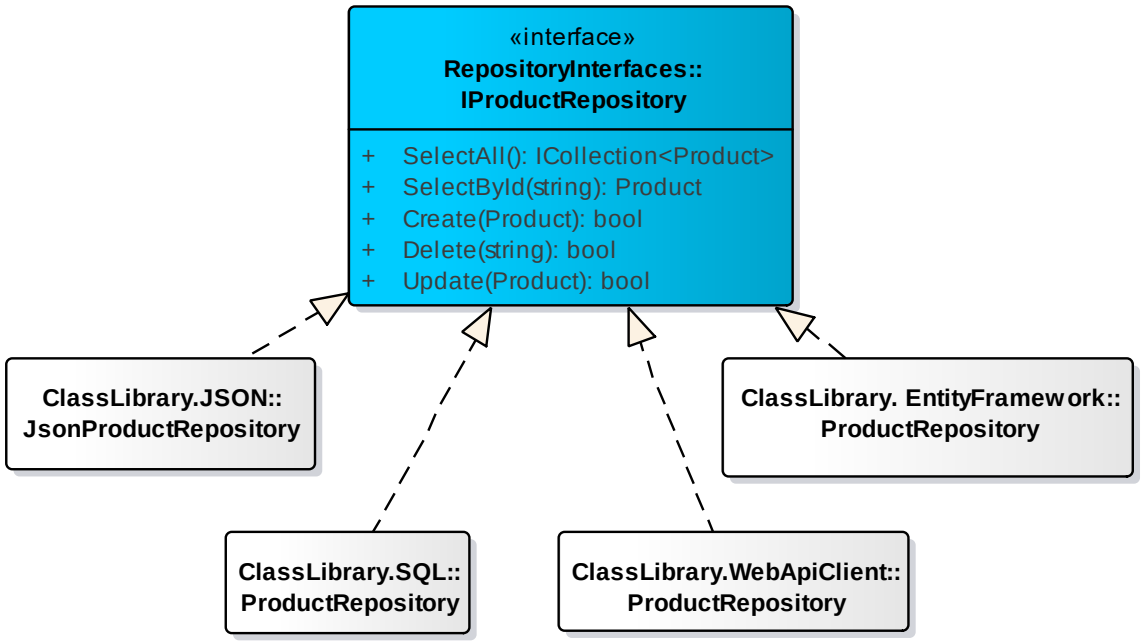
SqlProductRepository

.NET Standard class library

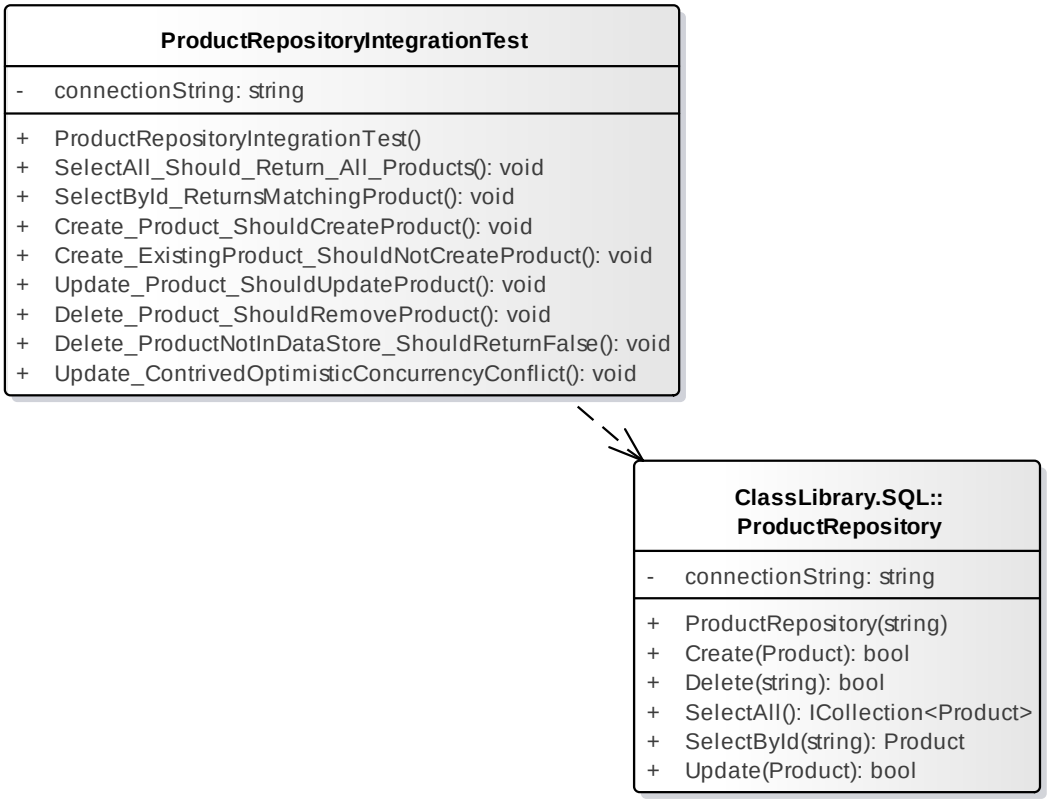
Add System.Data.SqlClient dependency using Nuget

Class	Namespace	Assembly		
		Core	.NET	Standard
String	System	System.Runtime	mscorlib	netstandard
List	System.Collections.Generic	System.Collections	mscorlib	netstandard
SqlConnection	System.Data.SqlClient	System.Data.SqlClient*	System.Data	System.Data.SqlClient*

*nuget



Integration Test



SQL

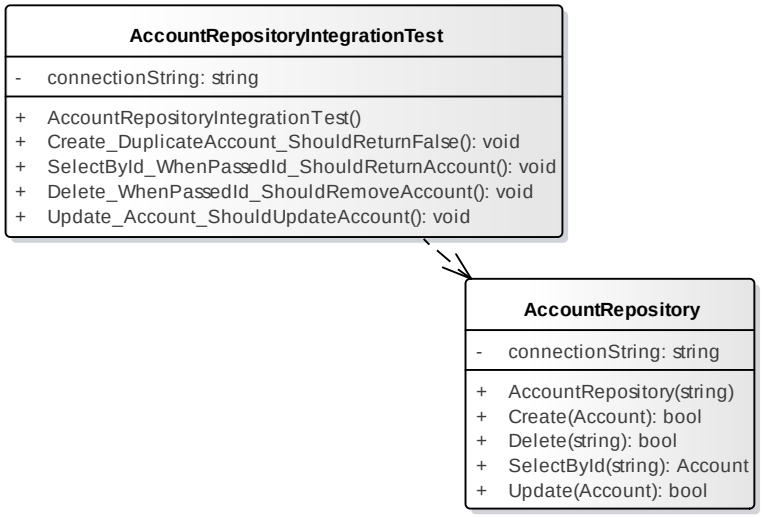
```
CREATE TABLE Products(
       Id nvarchar(100) NOT NULL,
       Name nvarchar(max) NOT NULL,
       CostPrice float NOT NULL,
       RetailPrice float NOT NULL,
       [RowVersion] rowversion,
       PRIMARY KEY(Id)
);
```

The Rowversion data type generates unique binary numbers for version-stamping table rows

Indexers

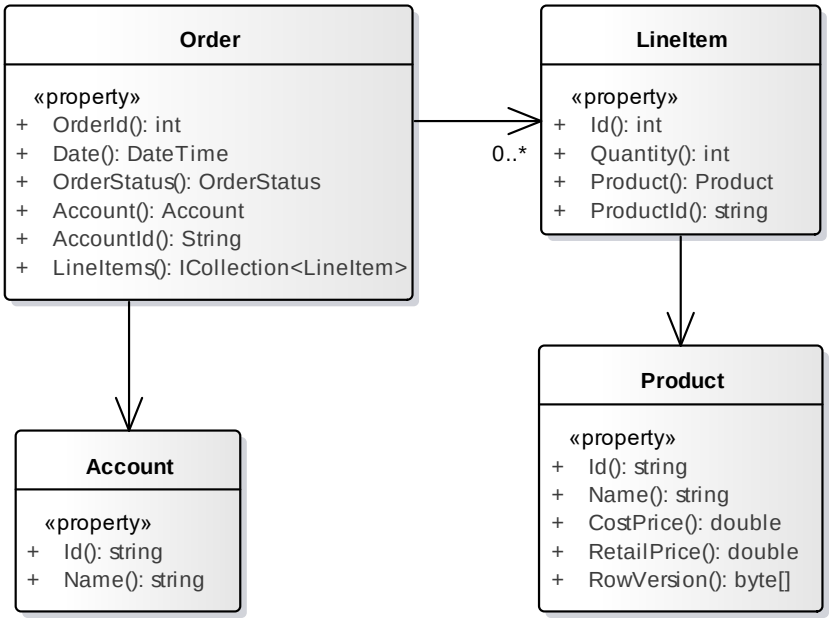
```
    public class ProductRepository : IProductRepository
    {
        //read only indexer implemented as expression-bodied member
        public Product this[string index]
        {
             get => SelectById(index);
        }
```

SqlAccountRepository



SqlOrderRepository

Class Diagram



Database tables

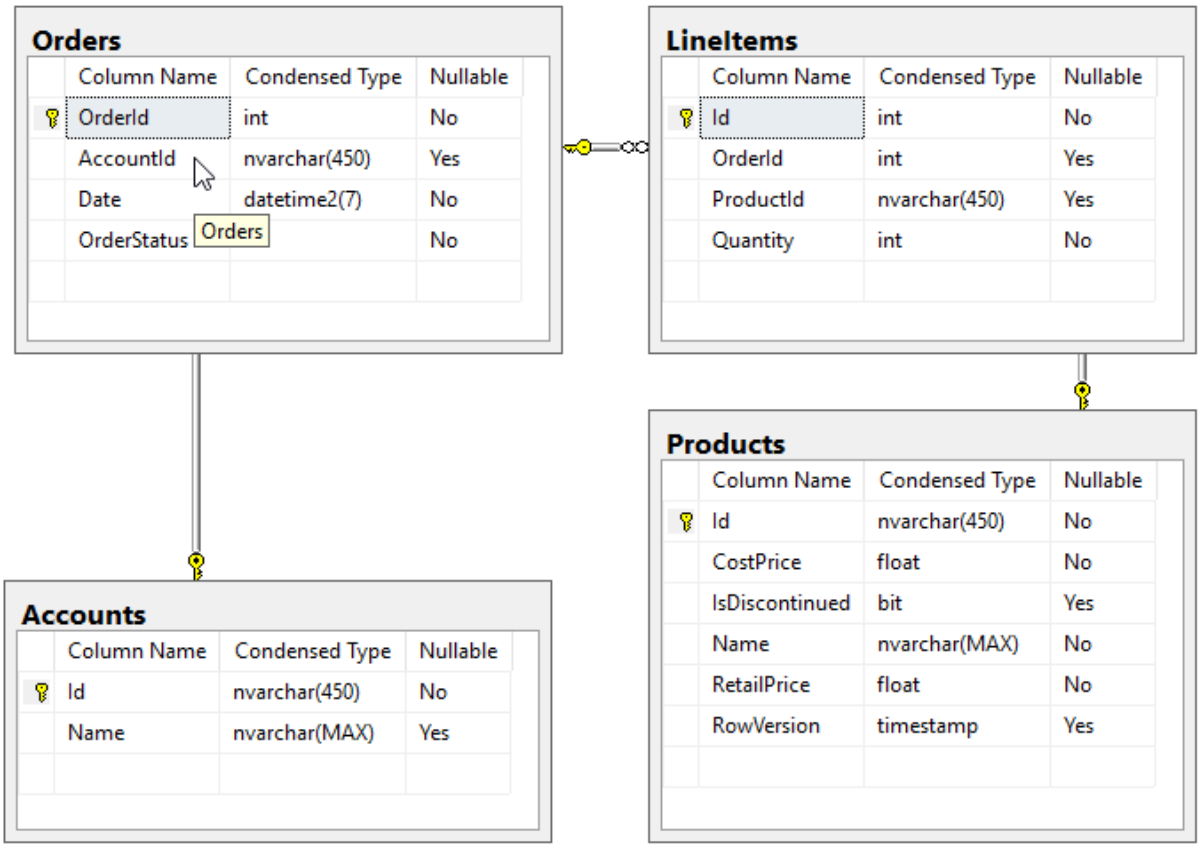
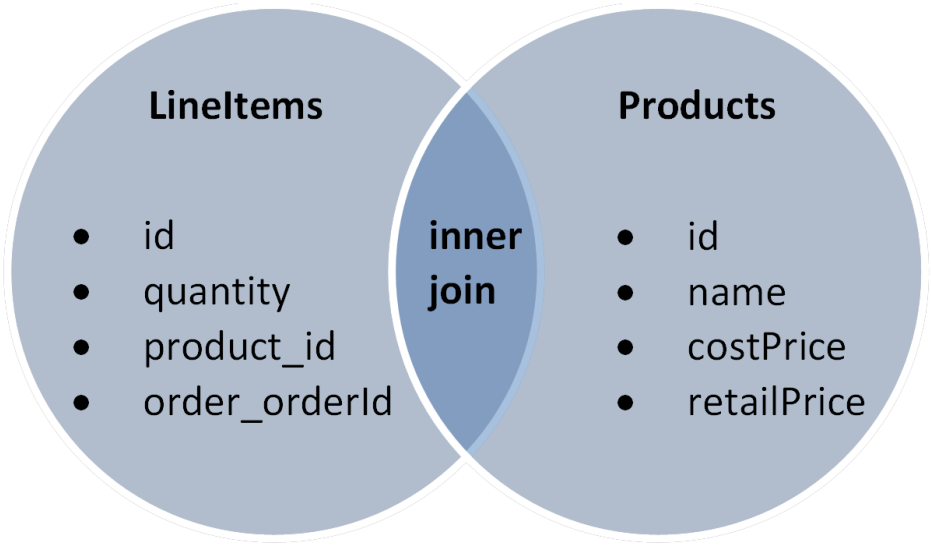


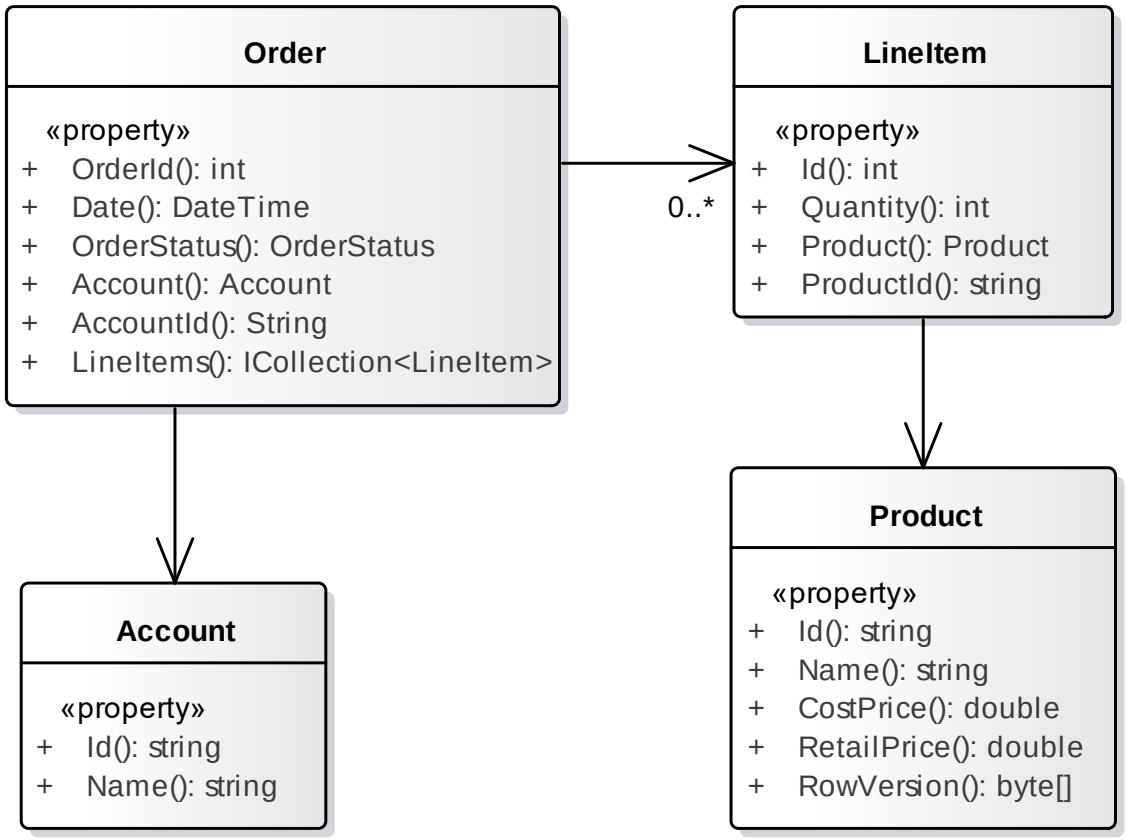
Table joins




```
select LineItems.Id, Products.Name from Products
inner join LineItems on LineItems.ProductId = Products.Id
where LineItems.OrderId = 1;
```

Entity Framework Core

EF Core is an object-relational mapper



If an entity has a navigation property, such as the Account property in the Order class, a foreign key property isn't required: EF automatically creates foreign keys in the database when they are needed. However, having the foreign key in the data model can make updates simpler and more efficient. For example, when retrieving an Order entity, the Account entity wil be null, but the foreign key property, AccountId will be set.

DbContext

- Used to query and save entity instances
- Derive from DbContext
- DbSet<TEntity> properties are automatically initialized
- Nuget entityframeworkcore.sqlserver

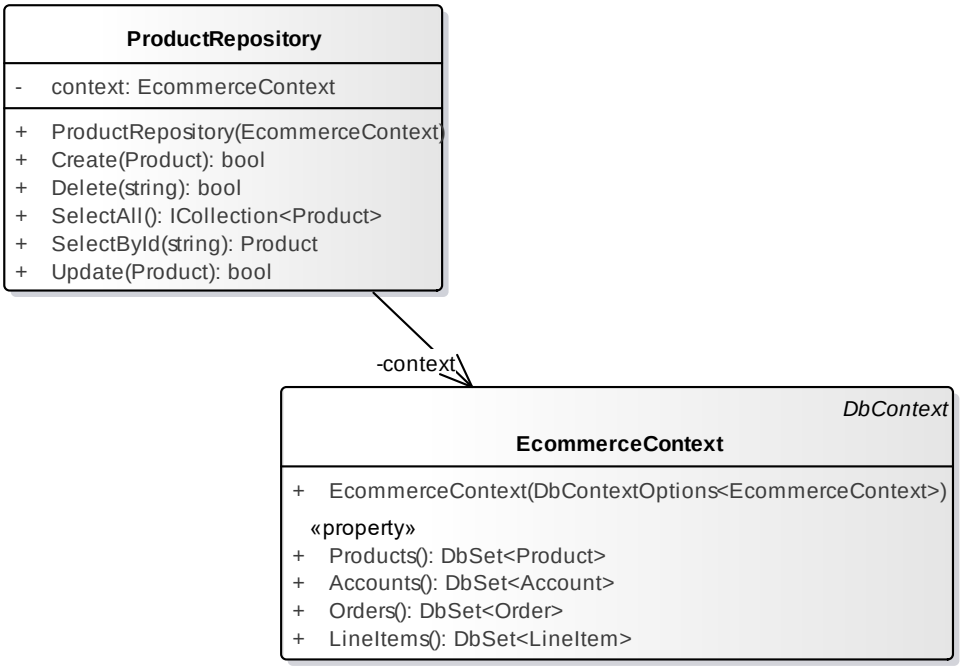
```
public class EcommerceContext : DbContext
{
    public EcommerceContext(DbContextOptions<EcommerceContext> options)
                                                            : base(options)
    {
    }
    public DbSet<Product> Products { get; set; }
    public DbSet<Account> Accounts { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<LineItem> LineItems { get; set; }
}
```

```
public class Product
{
    public string Id { get; set; }
    public string Name { get; set; }
    public double CostPrice { get; set; }
    public virtual double RetailPrice { get; set; }
    [Timestamp]
    public byte[] RowVersion { get; set; }
```

The Timestamp specifies the data type of the column as a row version. This attribute is in the Microsoft.EntityFrameworkCore assembly from Nuget.

ProductRepository

```
namespace ClassLibrary.EntityFramework
{
    public class ProductRepository : IProductRepository
    {
        private EcommerceContext context;

        public ProductRepository(EcommerceContext context)
        {
            this.context = context;
        }

        public bool Create(Product product)
        {
            context.Products.Add(product);
            try
            {
                return context.SaveChanges() == 1;
            }
            catch (DbUpdateException)
            {
                return false;
            }
        }

        public ICollection<Product> SelectAll()
        {
            return context.Products.ToList();
        }

        public Product SelectById(string id)
        {
            return context.Products.Find(id);
        }

        public bool Update(Product modifiedProduct)
        {
            Product product = context.Products.Find(modifiedProduct.Id);
            if (product == null)
            {
                return false;
            }
            product.CostPrice = modifiedProduct.CostPrice;
            product.RetailPrice = modifiedProduct.RetailPrice;
            return context.SaveChanges() == 1;
        }

        public bool Delete(string id)
        {
            Product product = context.Products.Find(id);
            if (product == null)
            {
                return false;
            }
            context.Remove(product);
            return context.SaveChanges() == 1;
        }

    }
}
```

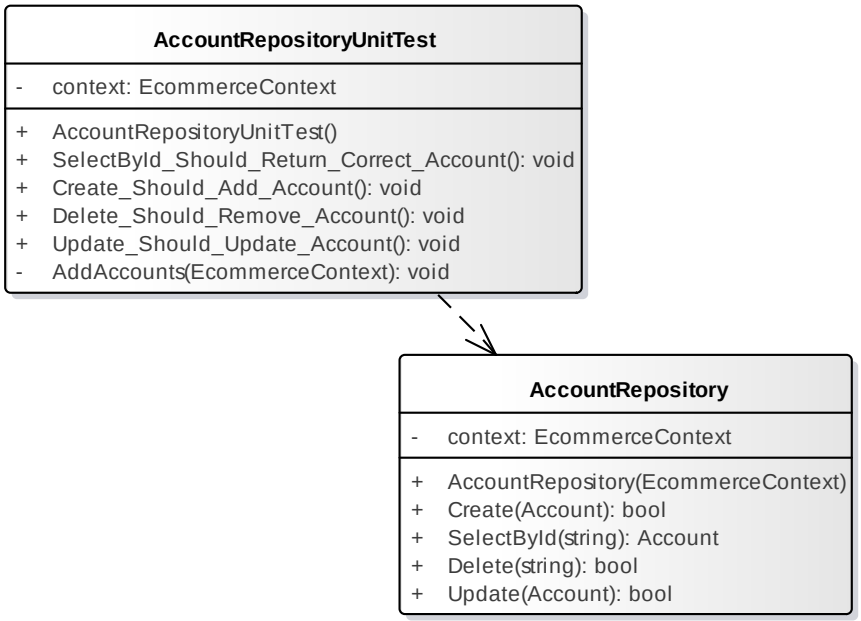


Unit Test with In Memory Database

```
namespace ClassLibrary.EntityFramework.Test
{
    public class ProductRepositoryUnitTest
    {
        protected EcommerceContext context;
        public ProductRepositoryUnitTest()
        {
            context = ContextFactory.InMemoryEcommerceContext;
        }

        ~ProductRepositoryUnitTest() => context.Dispose();

        [Fact]
        public void Create_Should_Add_Product()
        {
            //arrange
            Product product = new Product("p11", "Fridge", 100, 220);
            var productRepository = new ProductRepository(context);
            //act
            bool created = productRepository.Create(product);
            //assert
            Assert.Equal(1, context.Products.Count());
        }
```

The DbContext constructor takes a DbContextOptions argument, which can be obtained from DbContextOptionsBuilder.

```
// Microsoft.EntityFrameworkCore.InMemory Nuget package
namespace ClassLibrary.EntityFramework.Test
{
    public class ContextFactory
    {
        public static EcommerceContext InMemoryEcommerceContext
        {
            get
            {
                string dbname = Guid.NewGuid().ToString();
                var options = new DbContextOptionsBuilder<EcommerceContext>()
                                  .UseInMemoryDatabase(dbname)
                                  .Options;
                return new EcommerceContext(options);
            }
        }
```



```
namespace ClassLibrary.EntityFramework
{
    public class AccountRepository : IAccountRepository
    {
        private EcommerceContext context;

        public AccountRepository(EcommerceContext context)
        {
            this.context = context;
        }

        //Adding an Order with the Account property set
        //causes a row to be added to the the Accounts table
        public bool Create(Account account)
        {
            EntityEntry<Account> accountEntry = context.Accounts.Add(account);
            try {
                return context.SaveChanges() == 1;
            }
            catch (DbUpdateException)
            {
                return false;
            }
        }

        public Account SelectById(string id)
        {
            return context.Accounts.Find(id);
        }

        public bool Delete(string accountId)
        {
            Account account = context.Accounts.Find(accountId);
            if (account == null)
            {
                return false;
            }
            context.Remove(account);
            return context.SaveChanges() == 1;
        }

        public bool Update(Account modifiedAccount)
        {
            Account account = context.Accounts.Find(modifiedAccount.Id);
            if (account == null)
            {
                return false;
            }
            account.Name = modifiedAccount.Name;
            return context.SaveChanges() == 1;
        }
    }
}
```

SqlServer

Code First Migrations

Migrations enables changes to the model to be propagated to the database. The framework compares the current state of the model with the previous migration if one exists and generates a file containing a class inheriting from Migration with Up and Down methods.

Standard projects can't run code first migrations. Build a .NET Core Console App that references the Standard Class Library project.

Design-time services such as Migrations will automatically discover implementations of IDesignTimeDbContextFactory that are in the startup assembly or the same assembly as the derived context. By default, the migrations assembly is the assembly containing the DbContext. The following method sets the migrations assembly to the executing assembly.

```
// Factory for creating EcommerceContext
// entityframeworkcore.design Nuget pakage
public class EcommerceDesignTimeDbContextFactory :
                                        IDesignTimeDbContextFactory<EcommerceContext>
{
    public EcommerceContext CreateDbContext(string[] args)
    {
        string assemblyName = Assembly.GetExecutingAssembly().GetName().Name;
        string connectionString = "Data Source=.\\sqlexpress;Initial Catalog=db1;
                                        User ID=sa;Password=carpond";
        //By default the migrations assembly is the assembly containing the DbContext
        DbContextOptions<EcommerceContext> options = new
                                DbContextOptionsBuilder<EcommerceContext>()
                               .UseSqlServer(connectionString,
                                   b => b.MigrationsAssembly(assemblyName)).Options;
        EcommerceContext context = new EcommerceContext(options);
        return context;
    }
}
```

Package manager console commands

- Use Nuget to add entityframeworkcore.design pakage
- Set Startup project in Solution explorer and Default project in Package Manager Console to ConsoleApp

  `Add-Migration -Context EcommerceContext Initial`

  `Update-Database -Context EcommerceContext -Verbose`

Instead of including the connection string in the source code, it can be read from a json file. Set the file's Copy To Output Directory property to always

appsettings.json

```
{
  "ConnectionStrings": {
    "AuthenticationConnection": "Data Source=.\\sqlexpress;Initial
Catalog=authentication;User ID=sa;Password=carpond",
    "Db1Connection": "Data Source=.\\sqlexpress;Initial Catalog=db1;User
ID=sa;Password=carpond",
    "EcommerceConnection": "Data Source=.\\sqlexpress;Initial Catalog=ecommerce;User
ID=sa;Password=carpond"
  }
}
```

C#

```
string outputDirectory = AppDomain.CurrentDomain.BaseDirectory;
string connectionString = new ConfigurationBuilder()
           .SetBasePath(outputDirectory)
           .AddJsonFile("appsettings.json")
           .Build().GetConnectionString("EcommerceConnection");
```

CLI commands

1. In Solution Explorer, right-click the project and choose Open in File Explorer from the context menu.
2. Enter "cmd" in the address bar and press Enter.

```
> dotnet ef database drop
> dotnet ef migrations add -c EcommerceContext Initial
> dotnet ef database update -c EcommerceContext
```

Authentication tables

```
> dotnet ef migrations add -c ApplicationDbContext Initial
> dotnet ef database update -c ApplicationDbContext
```

Run from the Main method

```
namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {

            EcommerceContext context = new
                    EcommerceDesignTimeDbContextFactory().CreateDbContext(null);

            IProductRepository productRepository = new ProductRepository(context);
            ICollection<Product> products = productRepository.SelectAll();

            products.ToList().ForEach(p => Console.WriteLine(p.Name));

            Console.ReadKey();

        }
    }
}
```

To populate the table, run the SQL script in the Database project.

```
insert into products (id, [name], costprice, retailprice)
values ('p1','Dog''s Dinner', 0.70, 1.42) ;

insert into products (id, [name], costprice, retailprice)
values ('p2','Knife', 0.60, 1.20) ;

insert into products (id, [name], costprice, retailprice)
values ('p3','Fork', 0.55, 1.10) ;

insert into products (id, [name], costprice, retailprice)
values ('p4','Spaghetti', 0.44, 0.88) ;

insert into products (id, [name], costprice, retailprice)
values ('p5','Cheddar Cheese', 0.67, 1.34) ;

insert into products (id, [name], costprice, retailprice)
values ('p6','Bean bag', 11.20, 20.40) ;

insert into products (id, [name], costprice, retailprice)
values ('p7','Bookcase', 32, 64) ;

insert into products (id, [name], costprice, retailprice)
values ('p8','Table', 70, 140) ;

insert into products (id, [name], costprice, retailprice)
values ('p9','Chair', 60, 120) ;
```

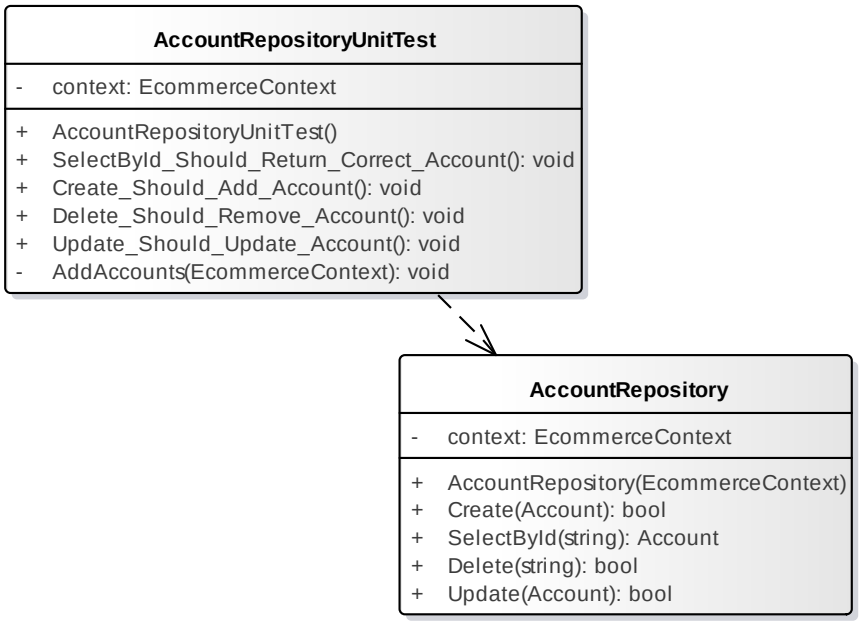
Integration Test

```
namespace ClassLibrary.EntityFramework.Test
{
    public class ContextFactory
    {
        public static EcommerceContext SqlServerEcommerceContext
        {
            get
            {
                string outputDirectory = AppDomain.CurrentDomain.BaseDirectory;
                string connectionString = new ConfigurationBuilder()
                    .SetBasePath(outputDirectory)
                    .AddJsonFile("appsettings.json")
                    .Build().GetConnectionString("EcommerceConnection");

                var options = new DbContextOptionsBuilder<EcommerceContext>()
                    .UseSqlServer(connectionString).Options;

                EcommerceContext context = new EcommerceContext(options);
                TruncateTables(context);
                return context;
            }
        }

        private static void TruncateTables(EcommerceContext context)
        {
            int rows = context.Database.ExecuteSqlCommand("delete from LineItems");
            rows = context.Database.ExecuteSqlCommand("delete from Orders");
            rows = context.Database.ExecuteSqlCommand("delete from Products");
            rows = context.Database.ExecuteSqlCommand("delete from Accounts");
            context.Database.ExecuteSqlCommand(
                                        "DBCC CHECKIDENT('Orders', RESEED, 0)");
            context.Database.ExecuteSqlCommand(
                                        "DBCC CHECKIDENT('LineItems', RESEED, 0)");
        }
    }
}
```

appsettings.json in ClassLibrary.EntityFramework project

```
{
  "ConnectionStrings": {
    "EcommerceConnection": "Data Source=.\\sqlexpress;Initial Catalog=ecommerce;
                            User ID=sa;Password=carpond"
  }
}

//select CopyToOutputDirectory
```

AccountRepository



Asynchronous methods

I/O Bound

- For I/O-bound code, await an operation which returns a Task or Task<T>
- await yields control to the caller of the method
- containing method declared async

```
namespace ClassLibrary.EntityFramework
{
    public class ProductRepository : IProductRepository, IProductRepositoryAsync
    {
        private EcommerceContext context;

        public ProductRepository(EcommerceContext context)
        {
            this.context = context;
        }

        public Product SelectById(string id)
        {
            return context.Products.Find(id);
        }

        public async Task<Product> SelectByIdAsync(string id)
        {
            Task <Product> task = context.Products.FindAsync(id);
            return await task;
        }
```

CPU Bound

- For CPU-bound code, await an operation which is started on a background thread with the Task.Run method.

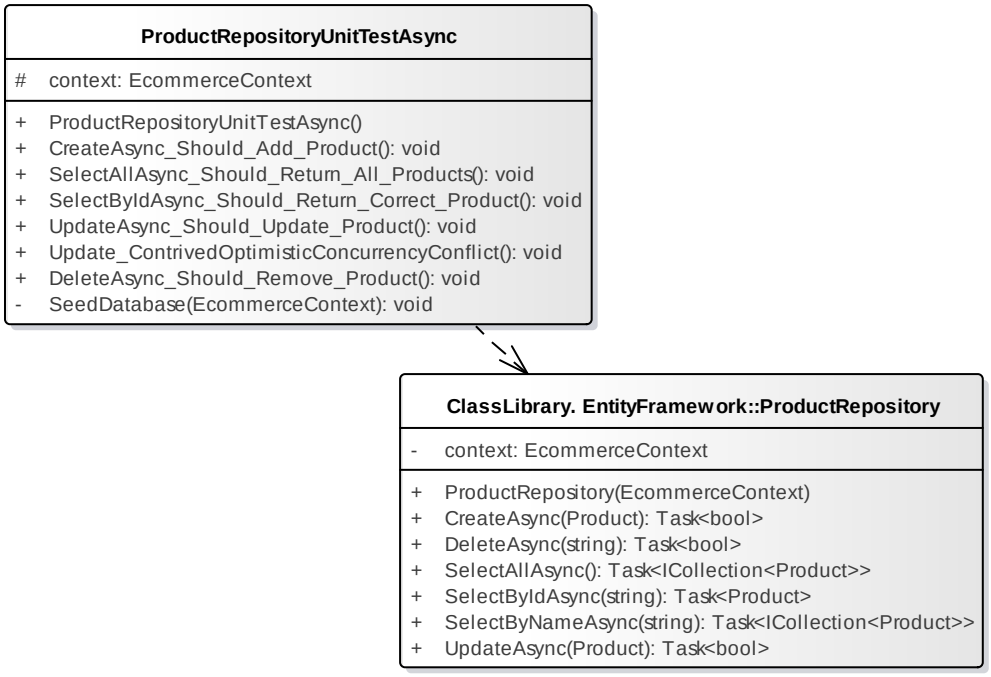
```
namespace ClassLibrary
{
    public class Maths
    {
        public async static Task<int> PrimeCountAsync(int max)
        {
            Func<int> func = () =>
                (from n in Enumerable.Range(2, max-1).AsParallel()
                 where Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)
                 select n).Count();
            int result = await Task.Run(func);
            return result;
        }
    }
}
```

Parameterized test

```
namespace ClassLibrary.Test
{
    public class MathsTest
    {
        [Theory]
        [InlineData(1e1, 4)]
        [InlineData(1e2, 25)]
        [InlineData(1e3, 168)]
        [InlineData(1e4, 1229)]
        [InlineData(1e5, 9592)]
        [InlineData(1e6, 78498)]
        [InlineData(1e7, 664579)]
        public async void PrimeCountAsyncTest(int limit, int count)
        {
            //act
            int actual = await Maths.PrimeCountAsync(limit);
            //assert
            Assert.Equal(count, actual);
        }
    }
}
```

Try it out

- Add duplicate async methods to ProductRepository
- Extract IAsyncProductRepository interface



OrderRepository

Interface

```
public interface IOrderRepositoryAsync
{
    // Create a new Order with OrderStatus set to Provisional
    Task<int> CreateAsync(Order order);

    IQueryable<Order> SelectAll();

    // Retrieve an Order with Provisional OrderStatus property
    Task<Order> SelectProvisionalOrderByAccountIdAsync(string accountId);

    Task<ICollection<Order>> SelectOrdersByAccountIdAsync(string accountId);
    Task<Order> SelectByOrderIdAsync(int orderId);
    Task<bool> UpdateAsync(Order order);
    Task<bool> DeleteAsync(int orderId);
}
```

Create an order

```
namespace ClassLibrary.EntityFramework
{
    public class OrderRepository : IOrderRepositoryAsync
    {
        public async Task<int> CreateAsync(Order order)
        {
            context.Add(order);
            await context.SaveChangesAsync();
            return order.OrderId;
        }
```

- If the order's Account property is set, a row will be added to the Account table
- Setting the order's foreign key property, AccountId, won't cause a cascade insert

Select Provisional Order

Eager Loading: related data is loaded from the database as part of the initial query

```
public async Task<Order> SelectProvisionalOrderByAccountIdAsync(
                                                    string accountId)
{
    Order order = await context.Orders
        .Include(o => o.Account)
        .Include(o => o.LineItems)
        .ThenInclude(li => li.Product)
        .FirstOrDefaultAsync(o => o.AccountId == accountId
        && o.OrderStatus == OrderStatus.Provisional);
    return order;
}
```

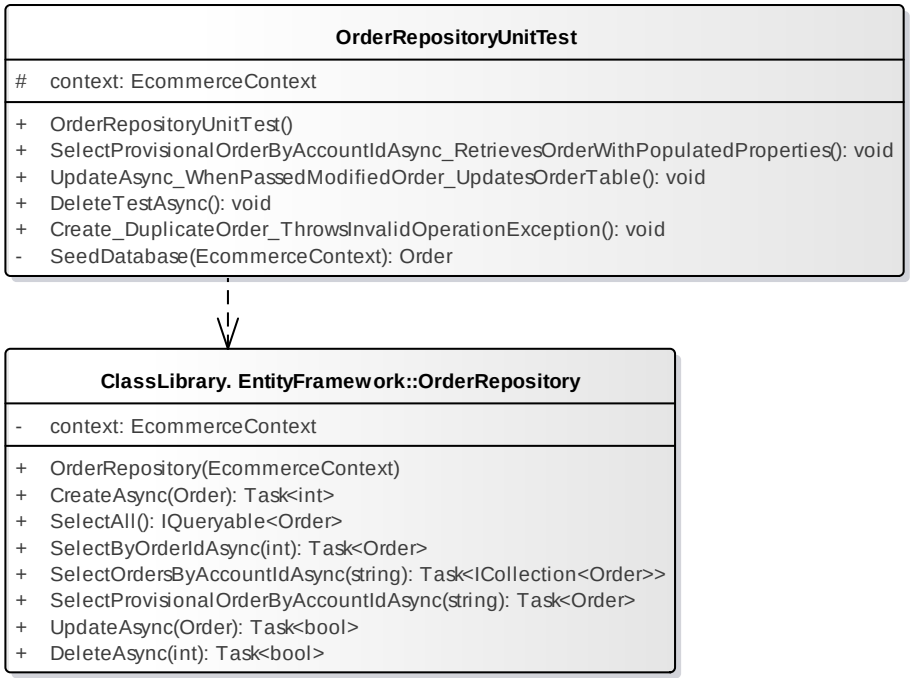
Update an Order

```
public async Task<bool> UpdateAsync(Order order)
{
    context.Entry(order).State = EntityState.Modified;
    int rowsUpdated = await context.SaveChangesAsync();
    return rowsUpdated == 1;
}
```

Entity State

Detached	The object exists but is not being tracked.
Added	The object has been added to the DbContext. SaveChanges has not been called.
Unchanged	The object has not been modified since it was attached to the context
Deleted	The object has been deleted from the DbContext.
Modified	One of the properties on the object was modified. SaveChanges has not been called.

Unit Tests

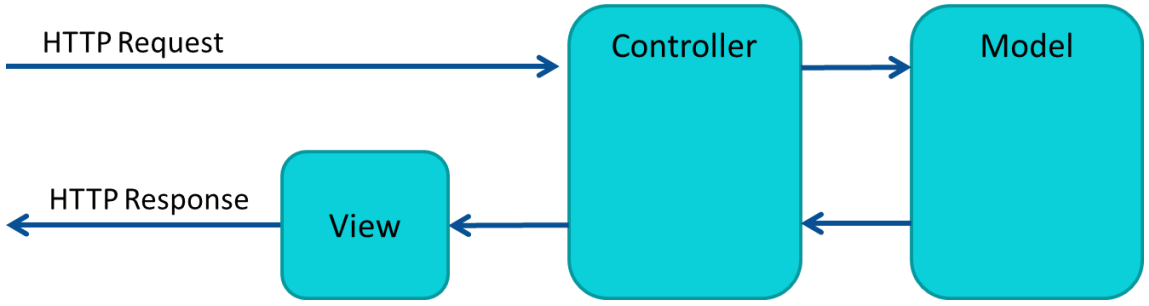


MVC Core

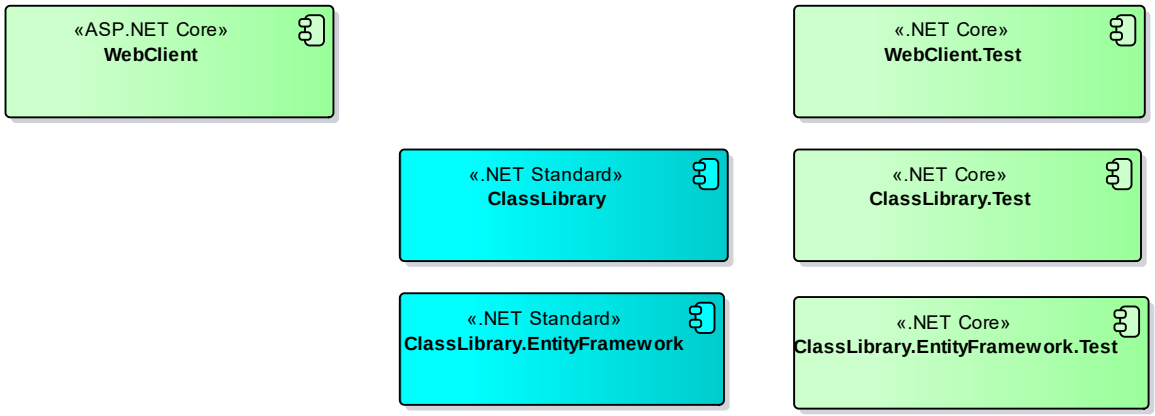
Overview

ASP.NET Core is a cross-platform open-source framework for building cloud-based applications. It comprises NuGet packages, so only the required dependencies can be included.

The model describes the data and business rules; the View is the application's user interface and the Controller handles communication with the user, overall application flow and application-specific logic.

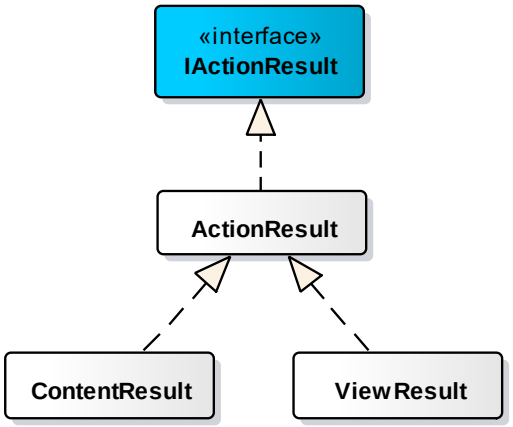


Add an ASP.NET Core MVC web application to the solution, and set the authentication option to 'Individual user Accounts'.



Controller

```
public class SimpleController : Controller
{
    public IActionResult Index()
    {
        return View(products);
    }

    public IActionResult Greet()
    {
        return Content("Hello");
    }
```



View

```
@using ClassLibrary.Entity;
@model ICollection<Product>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<body>
    @foreach(Product product in Model)
    {
        @product.Name
        <br/>
    }
</body>
</html>
```

```
public class SimpleController: Controller
{
    public IActionResult Index()
    {
        //Copy from Examples/Linq
        ICollection<Product> products = new List<Product> {
            new Product("p1", "Pedigree Chum", 0.70, 1.42),
            new Product("p2", "Knife", 0.60, 1.31),
            new Product("p3", "Fork", 0.75, 1.57),
            new Product("p4", "Spaghetti", 0.90, 1.92),
            new Product("p5", "Cheddar Cheese", 0.65, 1.47),
            new Product("p6", "Bean bag", 15.20, 32.20),
            new Product("p7", "Bookcase", 22.30, 46.32),
            new Product("p8", "Table", 55.20, 134.80),
            new Product("p9", "Chair", 43.70, 110.20),
            new Product("p10", "Doormat", 3.20, 7.40)
        };

        return View(products);
    }
}
```

RazorPage

Properties

- Model
- Layout
- User
- ViewBag

Methods

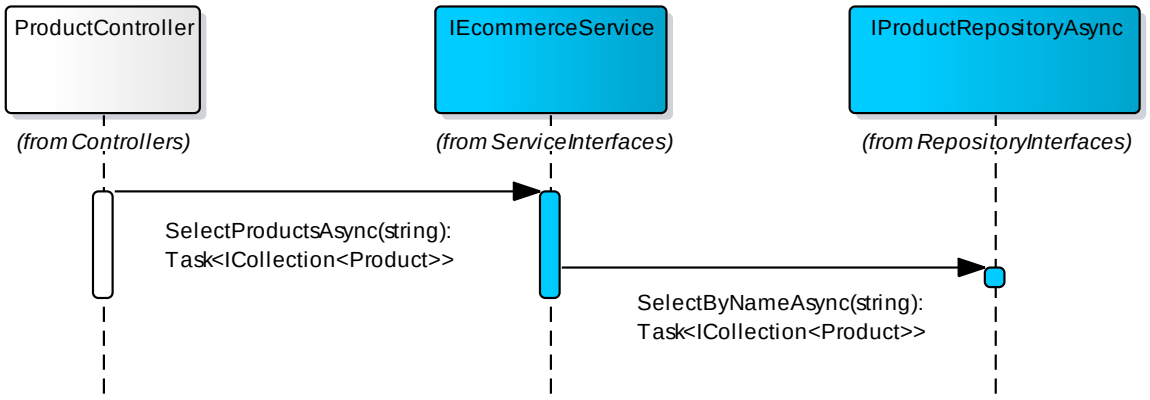
- RenderBody
- RenderSection

Keywords

- case
- do
- default
- for
- foreach
- if
- else
- not
- lock
- switch
- try
- catch
- finally
- using
- while

See documentation for RazorPage<TModel> Class and Razor reserved keywords

Service Layer



Separation of business logic (services) and data-access logic (repositories)

Interface

```
namespace ClassLibrary.ServiceInterfaces
{
    public interface IEcommerceService
    {
        // returns a selection of products or all products if argument is null
        Task<ICollection<Product>> SelectProductsAsync(string partOfName = null);

        // This will also add a row to the Account table if the Order's Account
        // Account property has been set
        Task<int> CreateOrderAsync(Order order);

        // add the product to the provisional order with the specified accountId
        Task<Order> AddProductToOrderAsync(string productId, string accountId);

        // Creates a new Order with the specified AccountId and with
        // OrderStatus = Provisional or returns an existing Provisional Order for the
        // AccountId
        Task<Order> CreateOrSelectProvisionalOrderForExistingAccount(
                                                                string accountId);

        // change the status of the provisional order with the specified
        // accountId to confirmed
        Task ConfirmOrderAsync(string accountId);

        // remove the product from the provisional order with the specified
        // accountId
        Task RemoveProductFromOrderAsync(string productId, string accountId);

        // Called from Create method of ProductController
        Task<bool> CreateProductAsync(Product product);
    }
}
```

Implementation

```
public class EcommerceService : IEcommerceServiceExtra
{
    private IProductRepositoryAsync productRepository;
    private IOrderRepositoryAsync orderRepository;

    public EcommerceService(IProductRepositoryAsync productRepository,
                                    IOrderRepositoryAsync orderRepository)
    {
        this.productRepository = productRepository;
        this.orderRepository = orderRepository;
    }

    public async Task<ICollection<Product>> SelectProductsAsync(
                                                    string partOfName = null)
    {
        ICollection<Product> products = partOfName == null ?
            await productRepository.SelectAllAsync() :
            await productRepository.SelectByNameAsync(partOfName);

        return products;
    }

   public async Task<int> CreateOrderAsync(Order order)
    {
        return await orderRepository.CreateAsync(order);
    }

    public async Task<Order> AddProductToOrderAsync(string productId,
                                                        string accountId)
    {
        Product product = await productRepository.SelectByIdAsync(productId);
        Order order = await
           orderRepository.SelectProvisionalOrderByAccountIdAsync(accountId);
        order.AddProduct(product, 1);
        await orderRepository.UpdateAsync(order);
        return order;
    }

    public async Task<Order> CreateOrSelectProvisionalOrderForExistingAccount(
                                                        string accountId)
    {
        Order order = await
           orderRepository.SelectProvisionalOrderByAccountIdAsync(accountId);
        if (order == null)
        {
            order = new Order();
            //setting AccountId foreign key property instead of Account
            //navigation property prevents cascade insert
            order.AccountId = accountId;
            int orderId = await orderRepository.CreateAsync(order);
        }
        return order;
    }
```

```
public async Task ConfirmOrderAsync(string accountId)
{
    Order order = await
       orderRepository.SelectProvisionalOrderByAccountIdAsync(accountId);
    if (order == null)
    {
        throw new InvalidOperationException(
            $"Provisional order for account {accountId} not found");
    }
    order.OrderStatus = OrderStatus.Confirmed;
    await orderRepository.UpdateAsync(order);
}

public async Task<bool> CreateProductAsync(Product product)
{
    return await productRepository.CreateAsync(product);
}

public async Task<Product> SelectProductByIdAsync(string id)
{
    return await productRepository.SelectByIdAsync(id);
}


public async Task RemoveProductFromOrderAsync(string productId,
                                              string accountId)
{
    Product product = await productRepository.SelectByIdAsync(productId);
    Order order =
      await orderRepository.SelectProvisionalOrderByAccountIdAsync(accountId);
    order.RemoveProduct(product, 1);
    await orderRepository.UpdateAsync(order);
}

public async Task<ICollection<Order>> SelectOrdersByAccountIdAsync(
                                                        string accountId)
{
    return await orderRepository.SelectOrdersByAccountIdAsync(accountId);
}

public async Task<Order> SelectOrderAsync(int orderId)
{
    return await orderRepository.SelectByOrderIdAsync(orderId);
}
```

Dependency injection

Dependency injection is a software design pattern that implements inversion of control for resolving dependencies. An injection is the passing of a dependency to a dependent. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

Benefits of DI include

- Maintainability
- Testability (using mock implementations of injected interfaces)
- Flexibility and Extensibility
- Loose Coupling (reducing the number of dependencies between the application's components)

Controller

Build a controller class, passing IEcommerceService into the constructor.

```
public class ProductController : Controller
{
    private IEcommerceService ecommerceService;
    public ProductController(IEcommerceService ecommerceService)
    {
        this.ecommerceService = ecommerceService;
    }
    // GET: Product
    public async Task<IActionResult> Index()
    {
        return View(await ecommerceService.SelectProductsAsync());
    }
```

ConfigureServices

Register the DbContext passed into the ProductRepository and OrderRepository constructors.

AddTransient<TService,TImplementation>(this IServiceCollection services,
Func<IServiceProvider,TImplementation> implementationFactory)

- Extension method of IServiceCollection
- Adds a transient service of the type specified in TService. A transient service is disposed after the request
- Implementation type specified in TImplementation
- Using the factory specified in implementationFactory. ctx is the IServiceProvider and ECommerceService is the TImplementation

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
       //additional configuration

       //add additional DbContext if separate security database
       services.AddEntityFrameworkSqlServer().AddDbContext<EcommerceContext>(
              options => options.UseSqlServer(
              Configuration.GetConnectionString("EcommerceConnection")));


       //specify implementation of IEcommerceService
       //services registered with AddTransient are disposed after the request
       services.AddTransient<IEcommerceService, EcommerceService>(ctx =>
       {
           EcommerceContext context = ctx.GetService<EcommerceContext>();
           return new EcommerceService(new ProductRepository(context),
                                       new OrderRepository(context));
       });

    }
```

Default Controller

```
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)

    {
        //additional configuration
        app.UseMvc(routes =>
        {
              routes.MapRoute(
              name: "default",
              template: "{controller=Product}/{action=Index}/{id?}");
        });
    }
```

Add appsettings.json to the project root

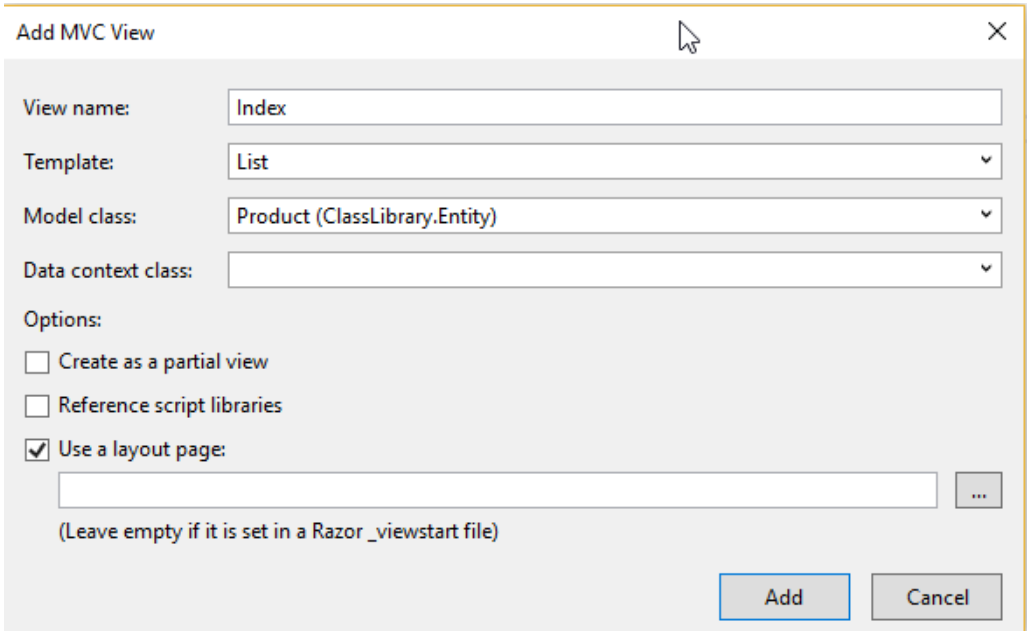
```
{
  "ConnectionStrings": {
    "AuthenticationConnection": "Data Source=.\\sqlexpress;Initial
              Catalog=authentication;User ID=sa;Password=carpond",
    "EcommerceConnection": "Data Source=.\\sqlexpress;Initial
              Catalog=ecommerce;User ID=sa;Password=carpond"
  }
}
```

Product List

Scaffolding

```
public class ProductController : Controller
{
    private IEcommerceService ecommerceService;
    public ProductController(IEcommerceService ecommerceService)
    {
        this.ecommerceService = ecommerceService;
    }
    // GET: Product
    public async Task<IActionResult> Index()
    {
        return View(await ecommerceService.SelectProductsAsync());
    }
```

Add a View using the List template



Tag Helpers

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.

```
<a asp-action="AddProduct"
asp-controller="Product"
asp-route-id="@item.Id">Select</a>
```


```
<a href="/Product/AddProduct/p1">Select</a>
```

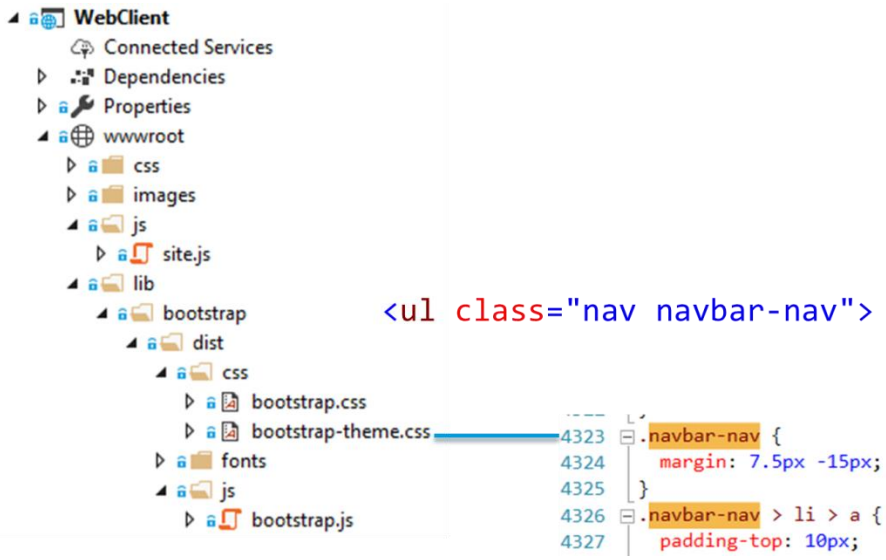
Built-in Tag Helpers

- Anchor
- Environment
- Form
- Image
- Input
- Label
- Select
- Textarea
- Validation Message
- Validation Summary

_Layout.cshtml

```
<ul class="nav navbar-nav">
<li><a asp-area="" asp-controller="Product" asp-action="Index">
Products</a></li>
<li><a asp-area="" asp-controller="Product" asp-action="Basket">
Basket</a></li>
<li><a asp-area="" asp-controller="Order" asp-action="Index">
Orders</a></li>
```

Bootstrap



_Layout.cshtml

```
<environment include="Development">
   <link rel="stylesheet"
         href="~/lib/bootstrap/dist/css/bootstrap.css" /> </environment>
<environment include="Production">
   <link rel="stylesheet"
         href="https://ajax.aspnetcdn.com/ajax/bootstrap/
                              3.3.7/css/bootstrap.min.css" </environment>
```

The environment can be configured in the project's debug profiles

ViewBag

This optionally uses a ViewBag dynamic property to retain the text in the input field

```
<form asp-action="Index">
    <input type="text" name="id" value="@ViewBag.SearchText" />
    <input type="submit" value="Search" class="btn btn-default" />
</form>
```

```
public class ProductController : Controller
{
    private IEcommerceService ecommerceService;
    public async Task<IActionResult> Index(string id = null)
    {
        ViewBag.SearchText = id;
        return View(await ecommerceService.SelectProductsAsync(id));
    }
```

ASP.NET Core Identity

The Identity services are added to the application in the ConfigureServices method in the Startup class. These services are made available to the application through dependency injection.

Identity is enabled for the application by calling UseAuthentication in the Configure method. UseAuthentication adds authentication middleware to the request pipeline.

Configure Security

Additional user properties

1. IdentityUser includes an Email property. Additional properties can be added; these will be stored in the security database.

```
namespace WebApplication1.Models
{
    public class ApplicationUser : IdentityUser
    {
        public string Name { get; set; }
    }
```

2. Add a field to Register.cshtml in Views/Account

```
<div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" class="form-control" />
    <span asp-validation-for="Name" class="text-danger"></span>
</div>
```

3. Add the Name property to the RegisterViewModel class in Models/AccountViewModels. The ViewModel is used to pass values between the Register View and the AccountController

```
public class RegisterViewModel
{
    [Required]
    public string Name { get; set; }
```

Creating the database

1. Edit appsettings.json. The string named DefaultConnection is used for security

```
{
  "ConnectionStrings": {
      "DefaultConnection": "Data Source=.\\sqlexpress;Initial
              Catalog=authentication;User ID=sa;Password=carpond",
      "EcommerceConnection": "Data Source=.\\sqlexpress;Initial
              Catalog=ecommerce;User ID=sa;Password=carpond"
  }
}
```

2. The Startup class retrives the security database's connection string from appsettings.json. ApplicationDbContext derives from IdentityDbContext. This is a DbContext with DbSet properties including Users and Roles.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
                 options.UseSqlServer(
                 Configuration.GetConnectionString("DefaultConnection")));

        services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
                   .AddDefaultTokenProviders();
```

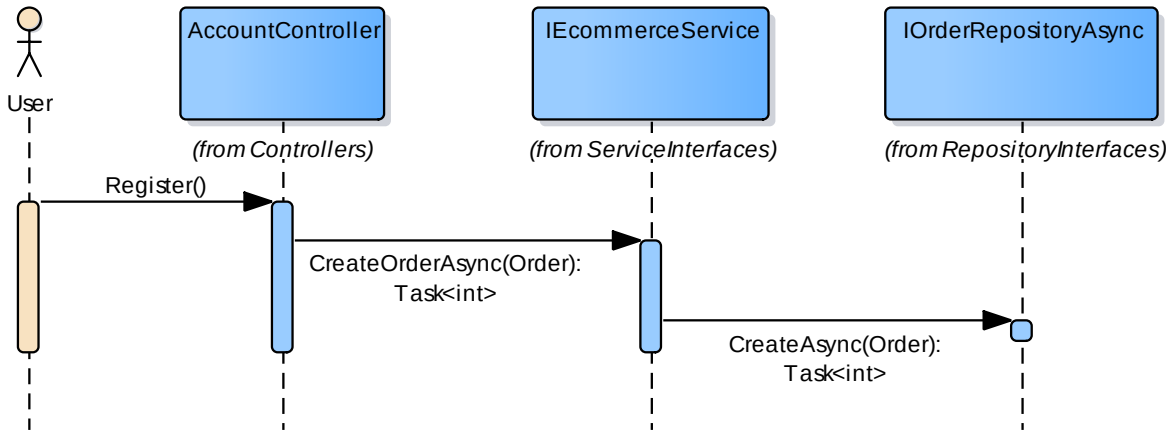
3. Create a database named authentication, set the default project in the package manager console to the web application and run migrations. Since there are two DbContext classes, the –Context parameter is included

```
Add-Migration -Context ApplicationDbContext Initial

Update-Database -Context ApplicationDbContext -Verbose
```

Password requirements

```
Optionally, add password requirements to Startup/ConfigureServices

    //password requirements
    services.Configure<IdentityOptions>(options =>
    {
           // Password settings
           options.Password.RequireDigit = false;
           options.Password.RequiredLength = 4;
           options.Password.RequireNonAlphanumeric = false;
           options.Password.RequireUppercase = false;
           options.Password.RequireLowercase = false;
           options.Password.RequiredUniqueChars = 1;
    });
```

Registering a new user

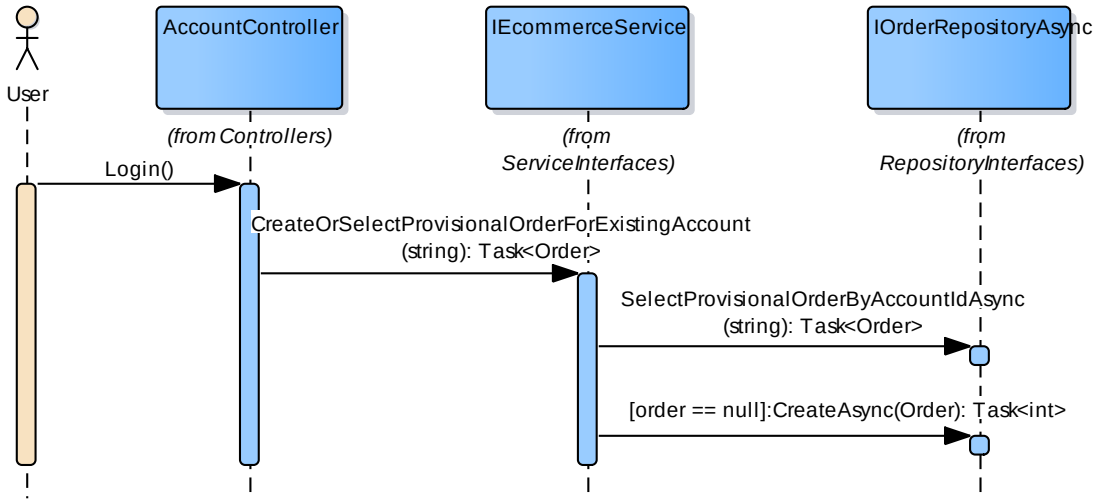


1. UserManager<ApplicationUser> and IEcommerceService are injected into the constructor
2. The Register method takes a RegisterViewModel argument and
    a. Calls the CreateAsync method of UserManager, which adds the user to the authentication table
    b. Builds an Order object, setting the Account and AccountId properties
    c. Calls the CreateOrderAsync method of IEcommerceService. This will add rows to the Order and Account tables

```
public class AccountController : Controller
{
    private readonly UserManager<ApplicationUser> _userManager;
    private IEcommerceService _ecommerceService;

    public AccountController(
        UserManager<ApplicationUser> userManager,
        IEcommerceService ecommerceService)
    {
        _userManager = userManager;
        _ecommerceService = ecommerceService;
    }

    [HttpPost]
    public async Task<IActionResult> Register(RegisterViewModel model,
                                                string returnUrl = null)
    {
        ViewData["ReturnUrl"] = returnUrl;
        if (ModelState.IsValid)
        {
            var user = new ApplicationUser { UserName = model.Email,
                                             Email = model.Email, Name=model.Name };
            var result = await _userManager.CreateAsync(user, model.Password);


            if (result.Succeeded)
            {
```

```
//setting Account property causes Account to be added to the
//database
Order order = new Order
{
    Account = new Account(model.Email, model.Name),
    AccountId = model.Email
};
int orderId = await _ecommerceService.CreateOrderAsync(order);

......

return RedirectToAction("Index", "Product");
```

Logging in




```
public class AccountController : Controller
{
  public async Task<IActionResult> Login(LoginViewModel model
  {
    if (result.Succeeded)
    {
        _logger.LogInformation("User logged in.");
        await _ecommerceService
           .CreateOrSelectProvisionalOrderForExistingAccountAsync (model.Email);
        return RedirectToAction("Index", "Product");
    }
```

Login.cshtml

During development, value attributes can be included in login page

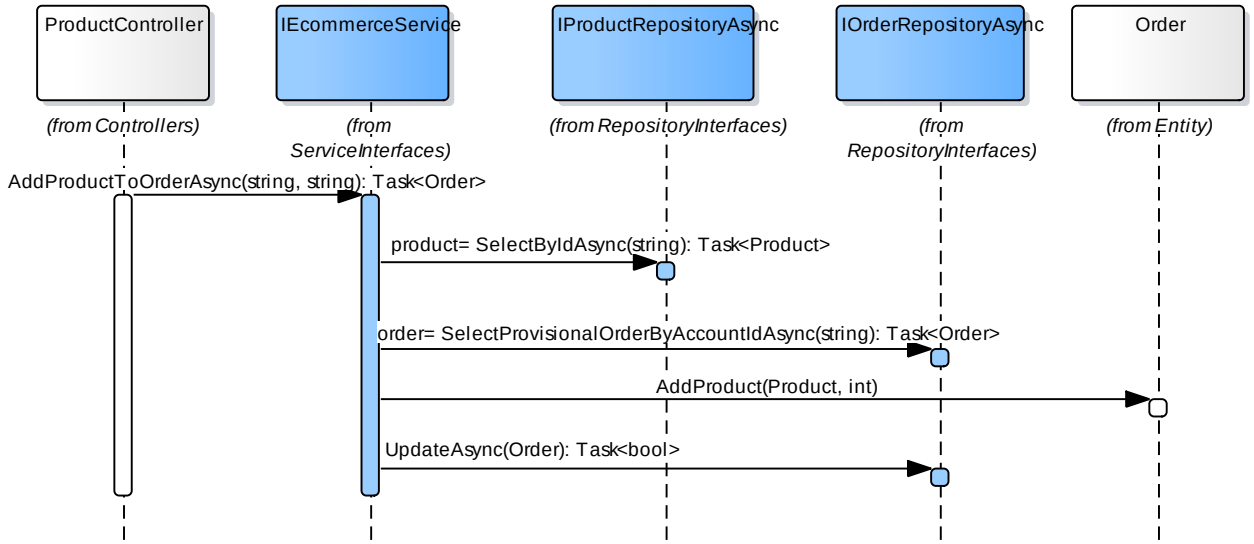
```
<form method="post"
    <div class="form-group">
        <label asp-for="Email"></label>
        <input asp-for="Email" class="form-control" value="a@b.com" />
```

```
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="Password"></label>
        <input asp-for="Password" class="form-control" value="delta1" />
        <span asp-validation-for="Password" class="text-danger"></span>
    </div>
```

Logging out

```
public async Task<IActionResult> LogOut() {
  await _signInManager.SignOutAsync();
  return RedirectToAction("Index", "Product");
```

Adding a product to the order

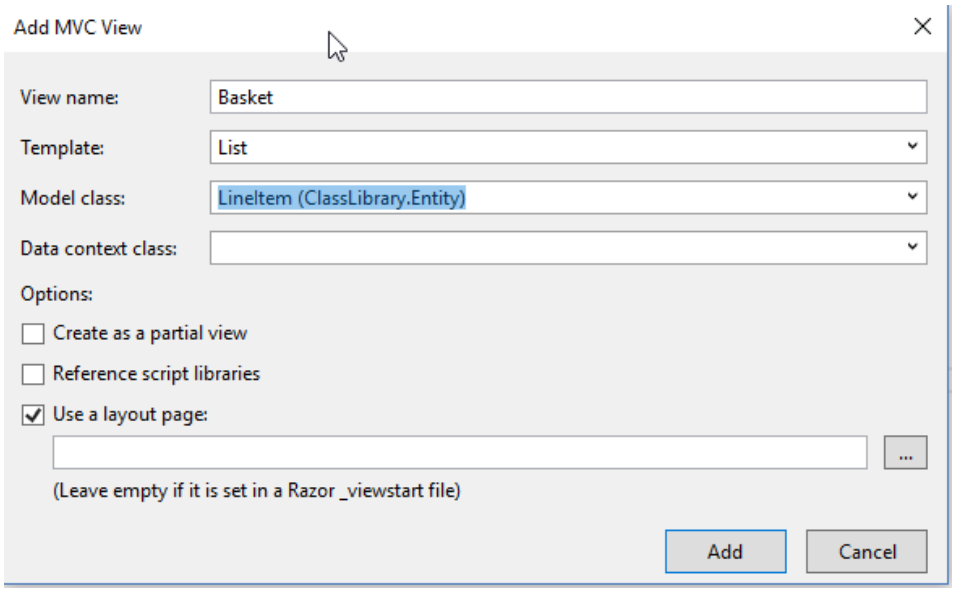



```
[Authorize]
public async Task<IActionResult> AddProduct(string id)
{
    Order order = await ecommerceService.AddProductToOrderAsync(id,
                                                                 User.Identity.Name);
    return View("Basket", order.LineItems);
}
```

`Task<Order> AddProductToOrderAsync(string productId, string accountId)`

1. retrieve Product from productRepository
2. retrieve Order from orderRepository
3. call AddProduct method of Order
4. call UpdateAsync method of orderRepository

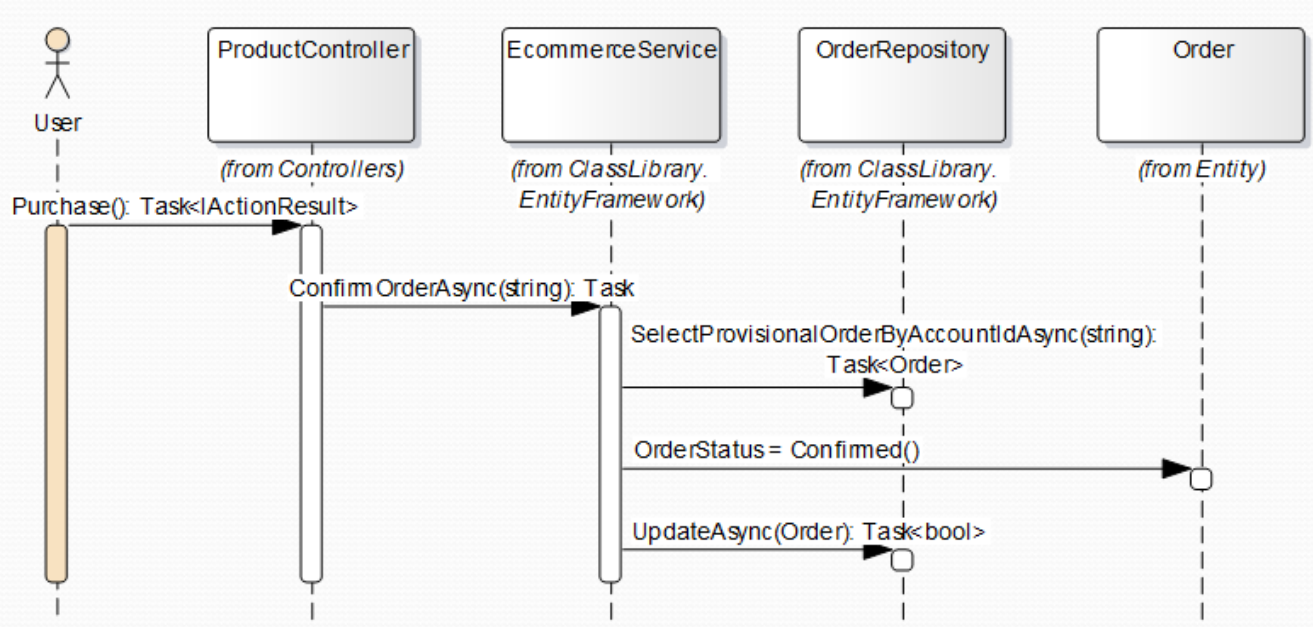
Basket View




```
@model IEnumerable<ClassLibrary.Entity.LineItem>

<table class="table">
    @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Product.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Quantity)
            </td>

            <td>
                <a asp-action="RemoveProductFromOrder"
                 asp-route-id="@item.Product.Id">Delete</a>
            </td>
        </tr>
    }
</table>
```

```
//GET: Product/Basket
[Authorize]
public async Task<IActionResult> Basket()
{
    Order order = await ecommerceService.
       CreateOrSelectProvisionalOrderForExistingAccountAsync(User.Identity.Name);
    return View(order.LineItems);
}
```

Purchase Action



Basket.cshtml

Either a link

```
<p>
    <a asp-action="Purchase">Purchase</a>
</p>
```

Or a button

```
<form asp-action="Purchase">
    <input type="submit" value="Purchase" class="btn btn-default" />
</form>
```

Controller

```
public class ProductController : Controller
{
    [Authorize]
    public async Task<IActionResult> Purchase()
    {
        await ecommerceService.ConfirmOrderAsync(User.Identity.Name);
        return View();
    }
```

Purchase.cshtml

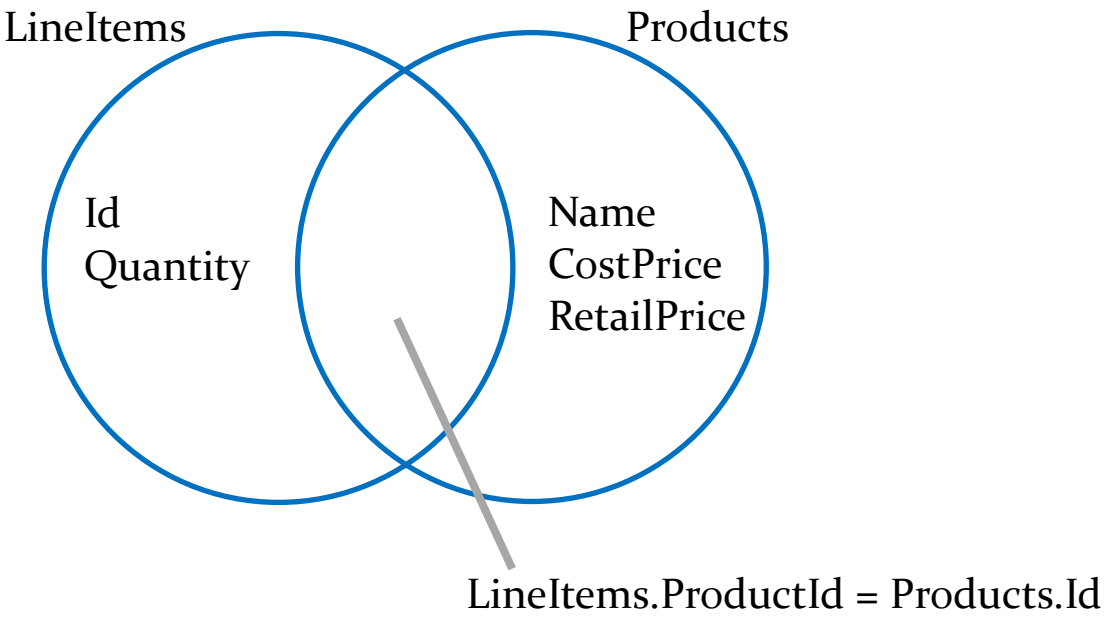
```
<h2>Purchase confirmed</h2>
```

Changing the Order Status

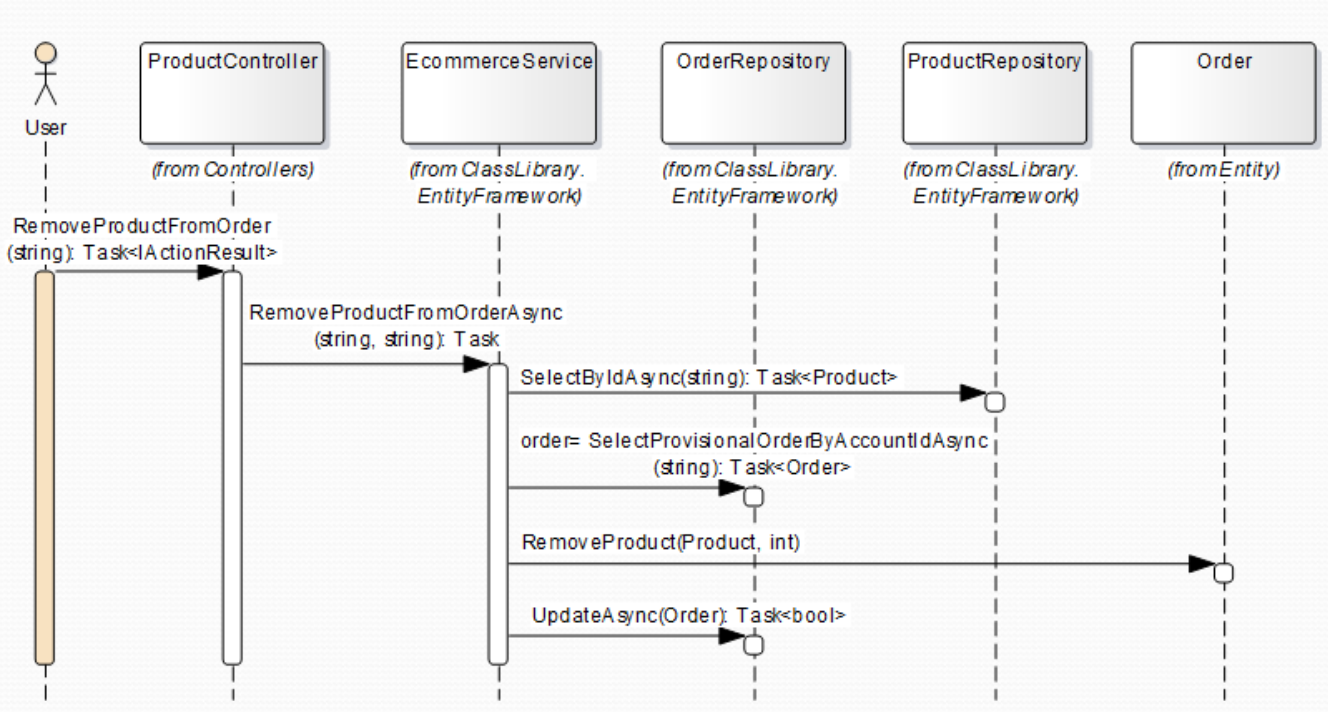
```
public class EcommerceService : IEcommerceService
{
    public async Task ConfirmOrderAsync(string accountId)
    {
        Order order = await orderRepository
               .SelectProvisionalOrderByAccountIdAsync(accountId);
        order.OrderStatus = OrderStatus.Confirmed;
        await orderRepository.UpdateAsync(order);
    }
```

Join.sql in Database project

```
DECLARE @id int
SET @id = 22

select orderId, id as [Account id], name from Accounts
inner join Orders on Orders.AccountId = Accounts.Id
where OrderId = @id;

select LineItems.Id as LineItemId, LineItems.Quantity, Products.Id as ProductId,
Products.Name, Products.CostPrice, Products.RetailPrice
from Products
inner join LineItems on LineItems.ProductId = Products.Id
where LineItems.OrderId = @id;
```



RemoveProductFromOrder



Basket.cshtml

```
<td>
    <a asp-action="RemoveProductFromOrder"
        asp-route-id="@item.Product.Id">Remove</a>
</td>
```

Controller

```
public class ProductController : Controller
{
    public async Task<IActionResult> RemoveProductFromOrder(string id)
    {
        await ecommerceService.RemoveProductFromOrderAsync(
                                                id, User.Identity.Name);
        return RedirectToAction(nameof(Basket));
    }
```

Service

```
namespace ClassLibrary.EntityFramework
{
    public class EcommerceService : IEcommerceService
    {
       public async Task RemoveProductFromOrderAsync(
                               string productId, string accountId)
        {
            Order order = await orderRepository
               .SelectProvisionalOrderByAccountIdAsync(accountId);
            Product product =
               await productRepository.SelectByIdAsync(productId);
            order.RemoveProduct(product, 1);
            await orderRepository.UpdateAsync(order);
        }
```

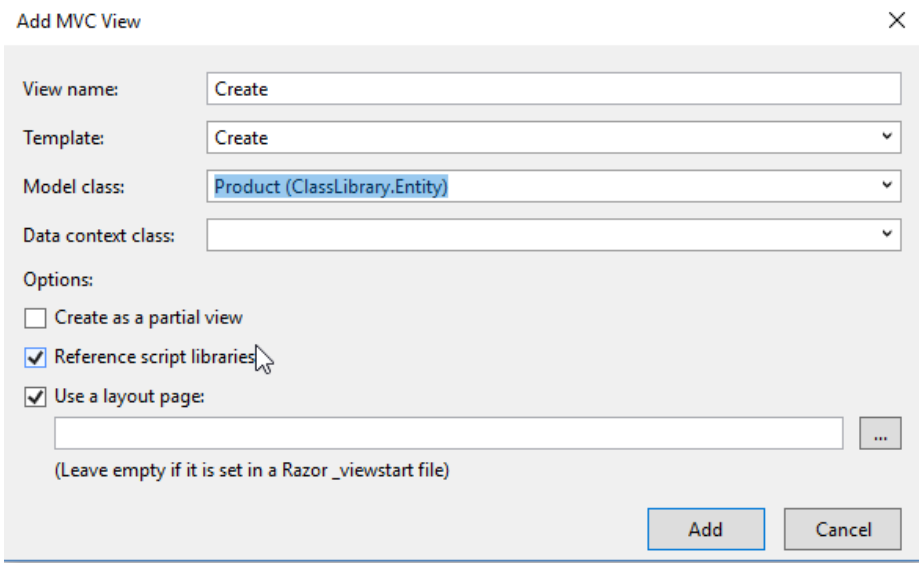
Order

```
namespace ClassLibrary.Entity
{
    public class Order
    {
        public bool RemoveProduct(Product product, int quantity)
        {
            LineItem lineItem = LineItems.FirstOrDefault(
                                       li => li.Product.Id == product.Id);
            if (lineItem == null)
            {
                throw new InvalidOperationException(
                                       $"{product.Name} is not in Order");
            }
            if (lineItem.Quantity >= quantity)
            {
                lineItem.Quantity -= quantity;
            }
            if (lineItem.Quantity == 0)
            {
                LineItems.Remove(lineItem);
            }
            if (lineItem.Quantity < quantity)
            {
                return false;
            }
            return true;
        }
```

Create

```
public class ProductController : Controller
{
    // GET: Product/Create
    public IActionResult Create()
    {
        return View();
    }
    // POST: Product/Create
    [HttpPost]
    public async Task<IActionResult> Create(Product product)
    {
        if (ModelState.IsValid)
        {
            await ecommerceService.CreateProductAsync(product);
            return RedirectToAction(nameof(Index));
        }
        return View(product);
    }
```

View



Tag helper

Tag helpers enable server-side code to generate HTML. The asp-for attribute value is a ModelExpression and the right hand side of a lambda expression. Id becomes m => m.Id in the generated code

```
<form asp-action="Create">
      <label asp-for="Id" />
      <input asp-for="Id" />
      <span asp-validation-for="Id" />
```

Generated HTML

```
<form action="/Product/Create" method="post">
        <div class="form-group">
              <label class="control-label" for="Id">Id</label>
              <input class="form-control" type="text" data-val="true"
                           data-val-required="Id is required." id="Id"
                           name="Id" value="" />
              <span class="text-danger field-validation-valid"
                           data-valmsg-for="Id"
                           data-valmsg-replace="true"></span>
        </div>
```

Prevent Cross-site request forgery

The FormTagHelper injects an anti-forgery token

```
<form action="/Product/Create" method="post">
        <input name="__RequestVerificationToken" type="hidden" value="8A7y6…"/>
</form>
```

Annotate controller methods with ValidateAntiForgeryToken. The Bind attribute prevents overposting.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create( [Bind("Id,Name,CostPrice,RetailPrice,
                                          RowVersion")] Product product)
{
```

jQuery validation

_Layout.cshtml

```
@RenderSection("Scripts", required: false)
```

Create.cshtml

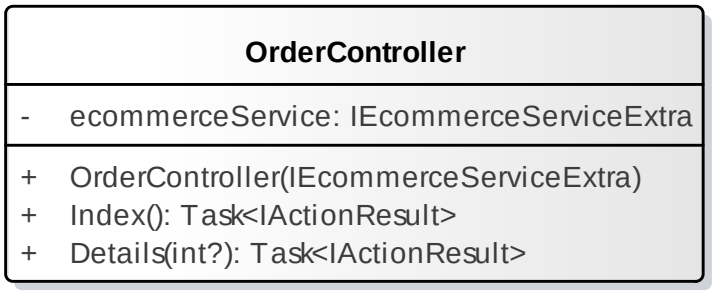
```
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

_ValidationScriptsPartial

```
<environment include="Development">
    <script src="~/lib/jquery-validation/dist/jquery.validate.js">
    </script>
    <script src="~/lib/jquery-validation-unobtrusive/
                                    jquery.validate.unobtrusive.js">
    </script>
</environment>
```

```
using System.ComponentModel.DataAnnotations;
namespace ClassLibrary.Entity
{
    public class Product
    {
        [Required(ErrorMessage = "Id is required.")]
        public string Id { get; set; }
        [RegularExpression(@"[a-zA-Z'\s]+", ErrorMessage = "Invalid")]
        [Required(ErrorMessage = "Name is required.")]
        public string Name { get; set; }
        [Range(0,1000)]
        [DisplayFormat(DataFormatString = "{0:C}")]
        public double CostPrice { get; set; }
        [Range(0, 2000)]
        [DisplayFormat(DataFormatString = "{0:C}")]
        public virtual double RetailPrice { get; set; }
        [Timestamp]
        [ScaffoldColumn(false)]
        public byte[] RowVersion { get; set; }
```

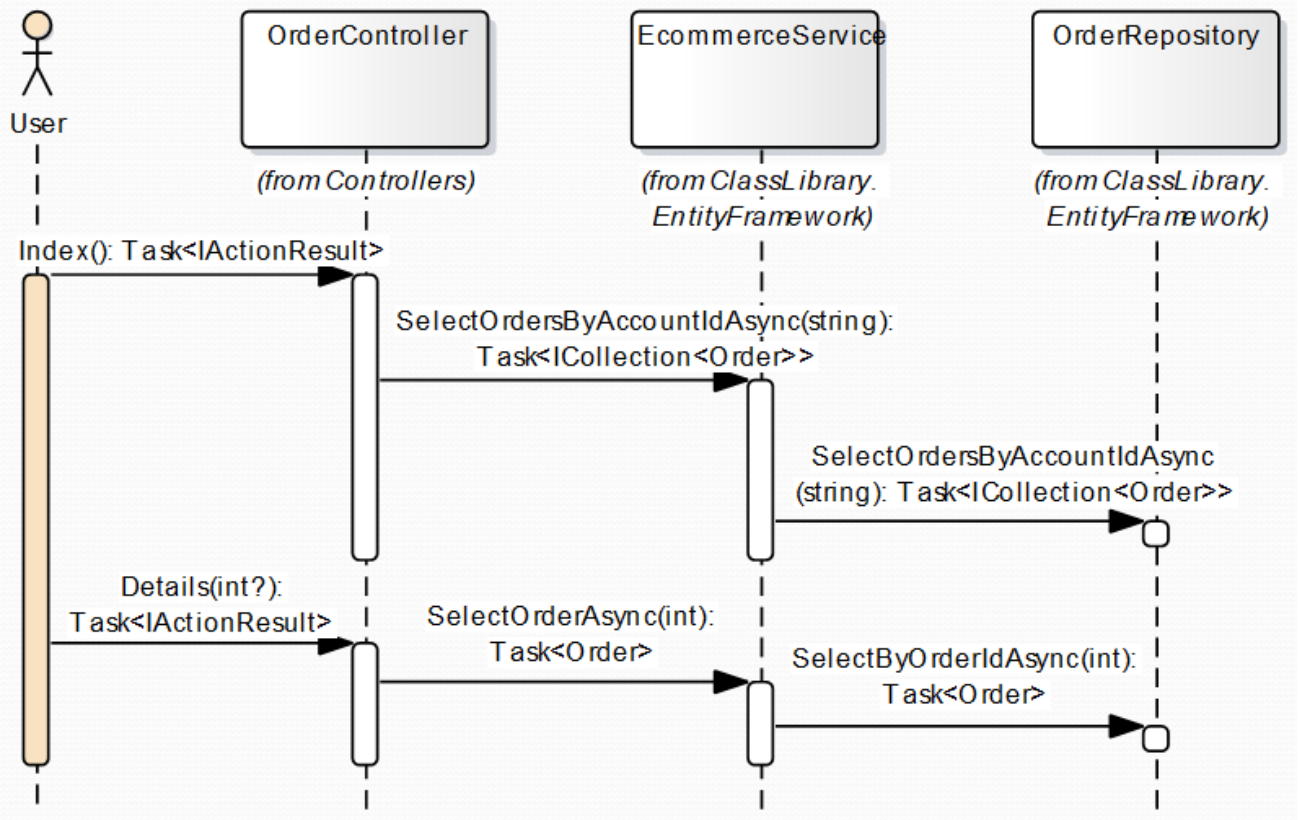
Orders



- Index displays a list of orders for the logged in account
- Details displays details of a selected order

_Layout.cshtml

```
<ul>
    <li><a asp-area="" asp-controller="Order" asp-action="Index">
        Orders</a>
    </li>
</ul>
```



OrderController

```
namespace WebClient.Controllers
{
    public class OrderController : Controller
    {
        // GET: Orders/
        [Authorize]
        public async Task<IActionResult> Index()
        {
            ICollection<Order> orders =
                await ecommerceService.SelectOrdersByAccountIdAsync(
                                                        User.Identity.Name);
            return View(orders);
        }

        // GET: Orders/Details/5
        [Authorize]
        public async Task<IActionResult> Details(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Order order = await ecommerceService.SelectOrderAsync((int)id);
            return View(order);
        }
```

EcommerceService

```
namespace ClassLibrary.EntityFramework
{
    public class EcommerceService : IEcommerceService
    {
        private IProductRepositoryAsync productRepository;
        private IOrderRepositoryAsync orderRepository;

        public async Task<ICollection<Order>> SelectOrdersByAccountIdAsync(
                                                                string accountId)
        {
            return await orderRepository.SelectOrdersByAccountIdAsync(accountId);
        }

        public async Task<Order> SelectOrderAsync(int orderId)
        {
            return await orderRepository.SelectByOrderIdAsync(orderId);
        }

    }

}
```

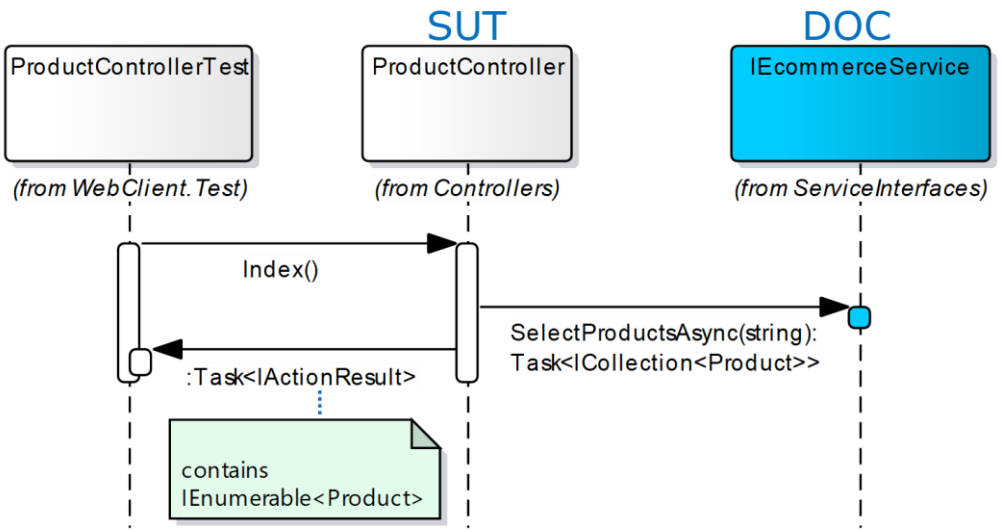
OrderRepository

```
namespace ClassLibrary.EntityFramework
{
    public class OrderRepository : IOrderRepositoryAsync
    {
        private EcommerceContext context;

        public OrderRepository(EcommerceContext context)
        {
            this.context = context;
        }

        //Eager loading
        public async Task<Order> SelectByOrderIdAsync(int orderId)
        {
            //Order order = await context.Orders.FirstOrDefaultAsync(
                                                    o => o.OrderId == orderId);

            return await context.Orders
                .Include(o => o.Account)
                .Include(o => o.LineItems)
                .ThenInclude(li => li.Product)
                .FirstOrDefaultAsync(o => o.OrderId == orderId);
        }

        public async Task<ICollection<Order>> SelectOrdersByAccountIdAsync(
                                                                string accountId)
        {
            return await context.Orders
                .Include(o => o.Account)
                .Include(o => o.LineItems)
                .ThenInclude(li => li.Product)
                .Where(o => o.AccountId == accountId).ToListAsync();
        }
    }
}
```

MVC Unit Tests

ProductController Test

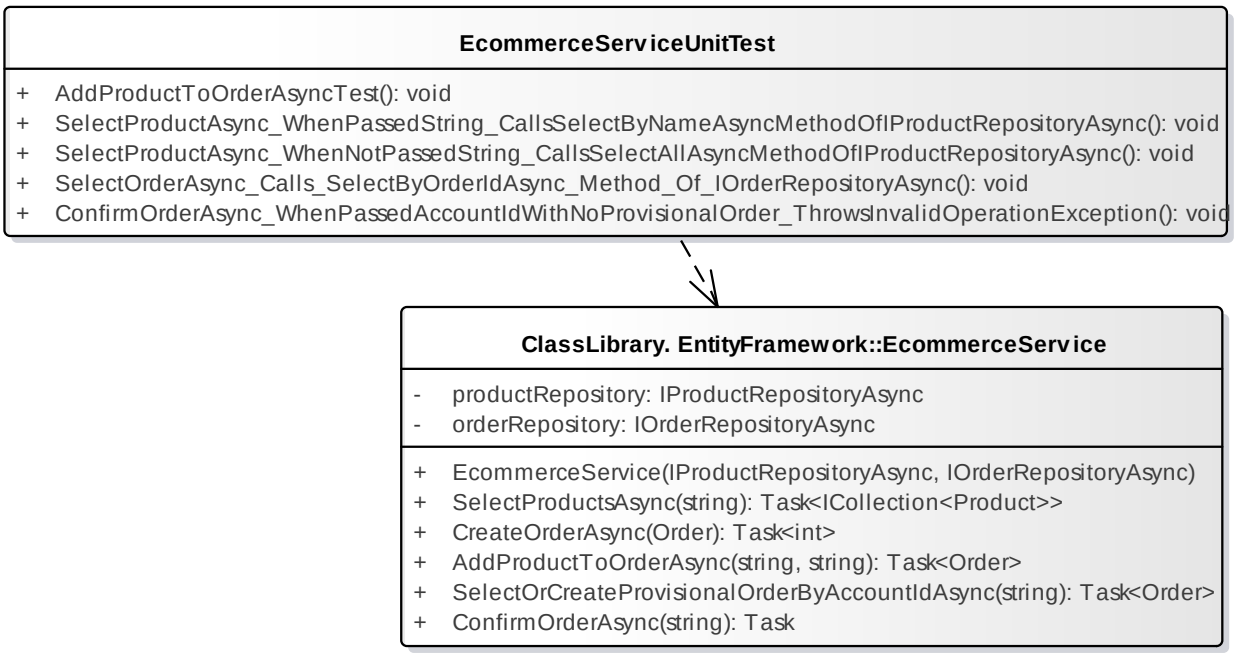
The Controller can be tested in isolation by passing a mock IEcommerceService instance into its constructor

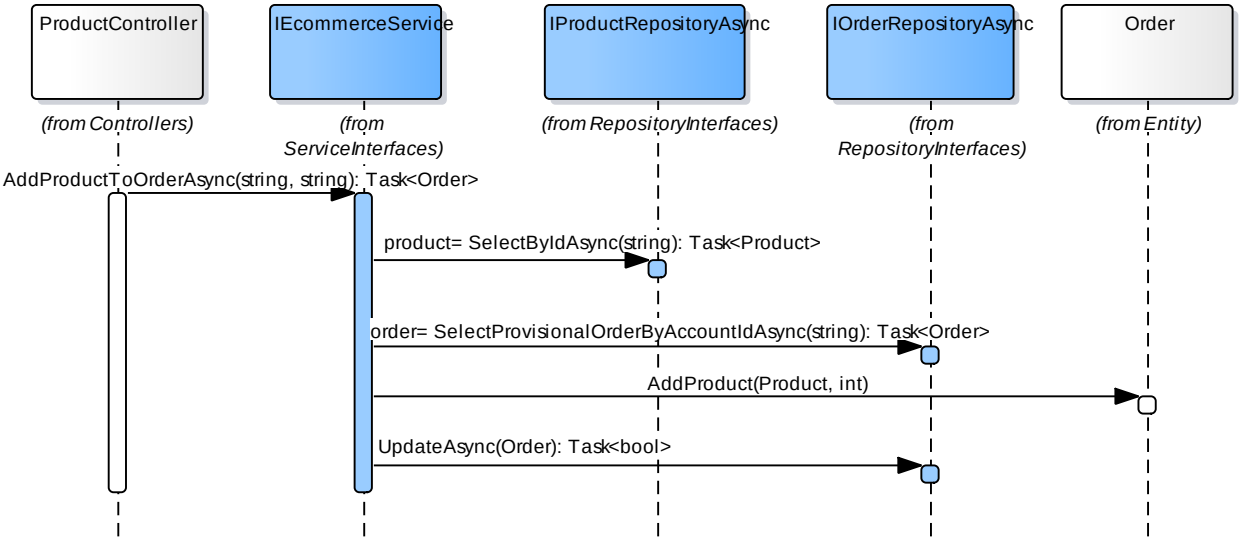



```
namespace WebClient.Test.Controllers
{
    public class ProductControllerTest
    {
        [Fact]
        public async Task Index_ReturnsAViewResult_WithACollectionOfProducts()
        {
            // Arrange
            Mock<IEcommerceServiceExtra> mockService =
                                                new Mock<IEcommerceServiceExtra>();
            mockService.Setup(svc => svc.SelectProductsAsync(null))
                                       .Returns(Task.FromResult(GetTestProducts()));
            ProductController controller = new ProductController(
                                                        mockService.Object, null);

            // Act
            ViewResult result = await controller.Index() as ViewResult;

            // Assert
            IEnumerable<Product> model =
                    Assert.IsAssignableFrom<IEnumerable<Product>>(result.Model);
            Assert.Equal(10, model.Count());
        }
```

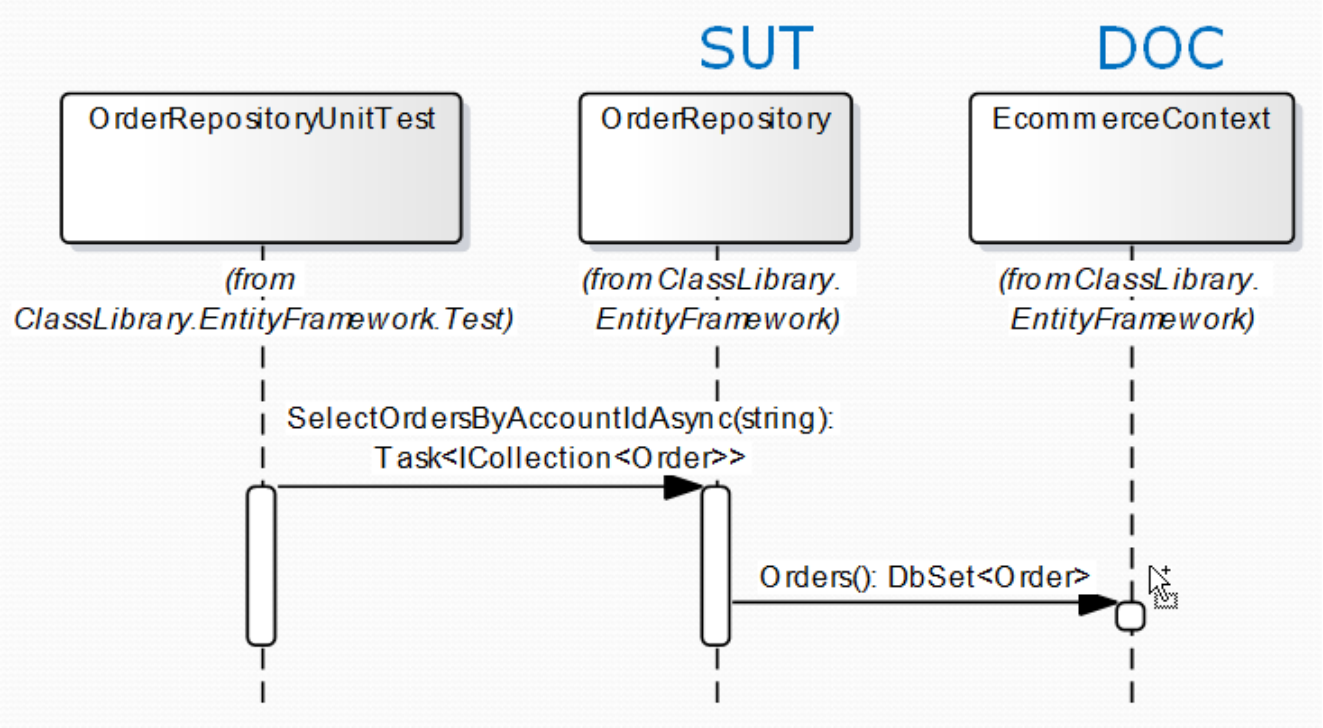
EcommerceService Test

Mock implementations of the Repository interfaces are passed into the constructor of the system under test



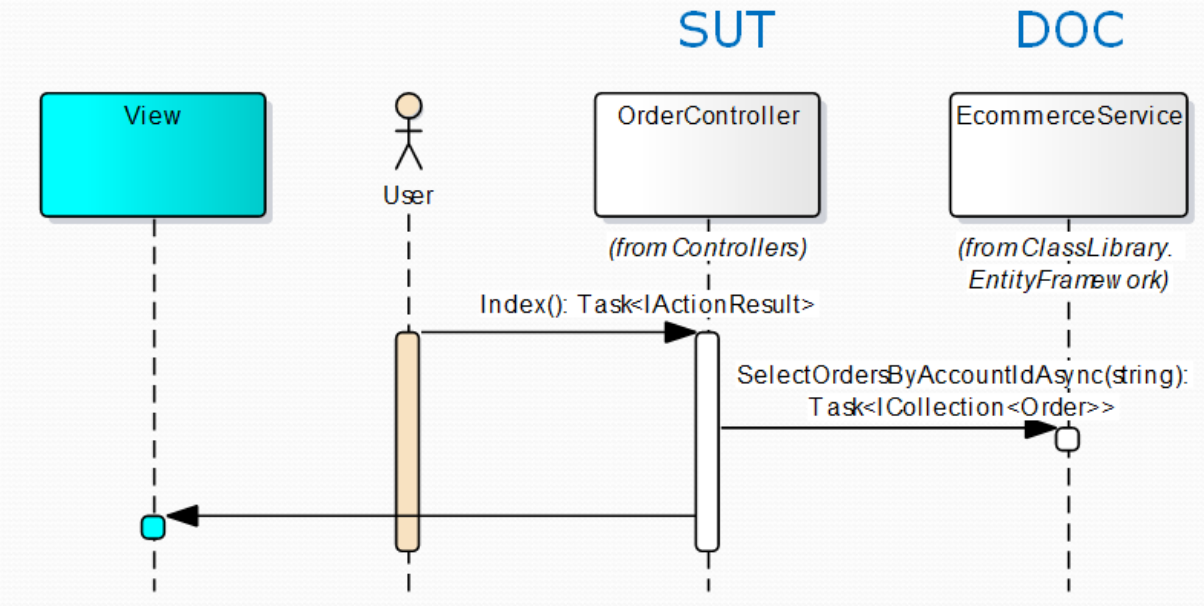


OrderRepository Test




```
namespace ClassLibrary.EntityFramework.Test
{
    public class OrderRepositoryUnitTest
    {
        [Fact]
        public async void SelectOrdersByAccountIdAsyncTest()
        {
            //arrange
            SeedDatabase(context);
            OrderRepository orderRepository = new OrderRepository(context);
            //act
            ICollection<Order> orders = await orderRepository
                                        .SelectOrdersByAccountIdAsync("acc1");
            //assert
            Assert.Equal("Pedigree Chum",
                                orders.First().LineItems.First().Product.Name);
        }
```

OrderController Test



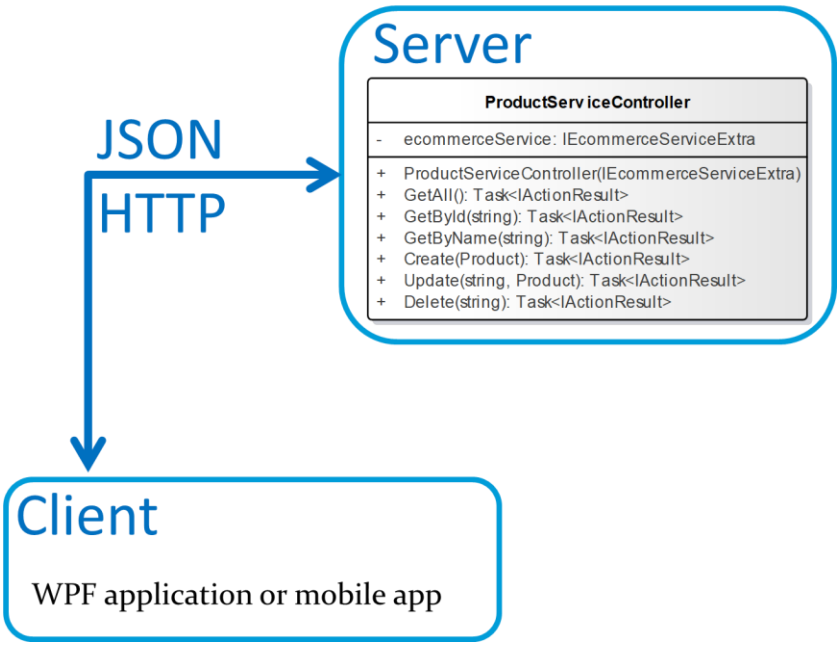
When calling the Index action from a unit test, the Controller's User property won't be initialised. To avoid a NullReferenceException, pass a mock IHttpContextAccessor into the controller's constructor.

```
namespace WebClient.Controllers
{
    public class OrderController : Controller
    {
        private IEcommerceServiceExtra ecommerceService;
        private IHttpContextAccessor context;
        public OrderController(IEcommerceService ecommerceService,
                                        IHttpContextAccessor context)
        {
            this.ecommerceService = ecommerceService;
            this.context = context;
        }
        [Authorize]
        public async Task<IActionResult> Index()
        {
            String username = context.HttpContext.User.Identity.Name;
```

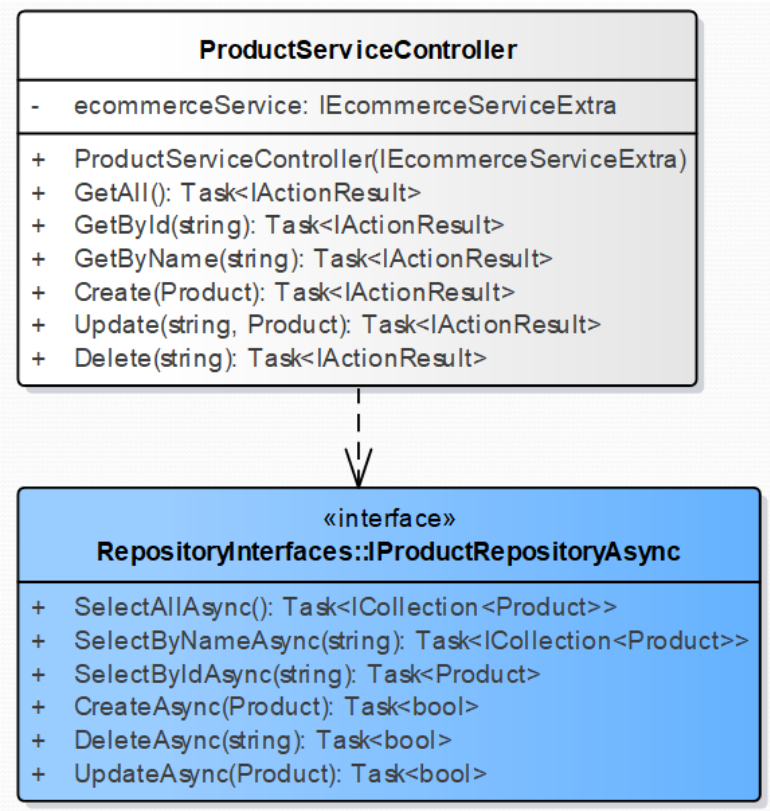
```
namespace WebClient.Test.Controllers
{
      public class OrderControllerTest
      {
        [Fact]
        public async Task Index_ReturnsAViewResult_WithACollectionOfOrders()
        {
            ICollection<Order> orders = new List<Order>();
            Mock<IEcommerceServiceExtra> mockService =
                                        new Mock<IEcommerceServiceExtra>();
            mockService.Setup(svc => svc.SelectOrdersByAccountIdAsync("user1"))
                                         .Returns(Task.FromResult(orders));
            Mock<IHttpContextAccessor> mockContext =
                                              new Mock<IHttpContextAccessor>();
            mockContext.Setup(c => c.HttpContext.User.Identity.Name)
                                                              .Returns("user1");
            OrderController controller = new OrderController(
                                    mockService.Object, mockContext.Object);
            // Act
            ViewResult result = await controller.Index() as ViewResult;
            // Assert
            Assert.IsAssignableFrom<ICollection<Order>>(result.Model);
        }
```

Web API

A framework for building HTTP REST services for clients including browsers and mobile devices.



ProductServiceController derives from Controller and contains methods whose annotatations correspond with HTTP methods. The controller connects to the repository layer to enable a client to store and retrieve products.



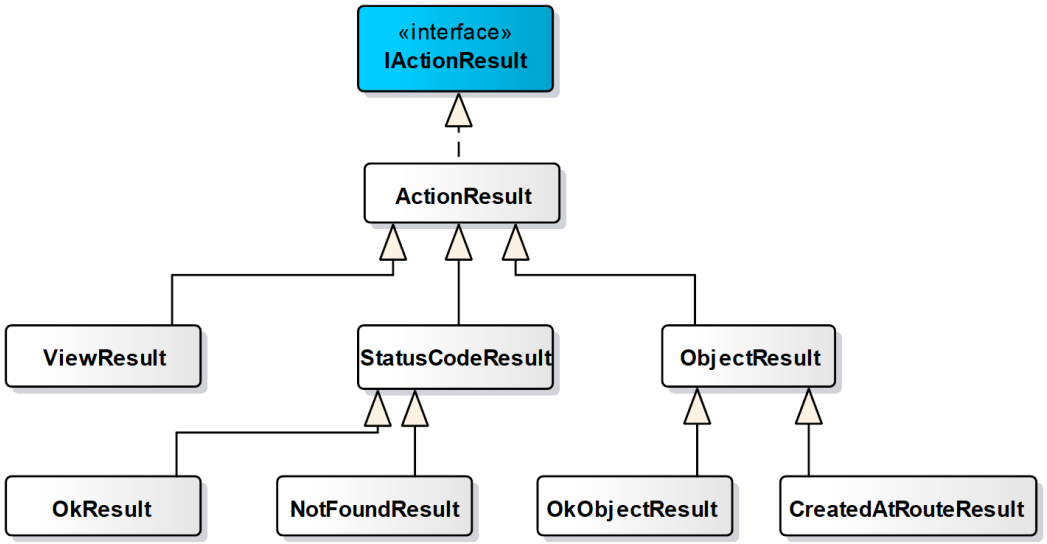
Route mapping

Add a Web API controller with read/write actions

```
namespace WebClient.Controllers
{
    [Route("api/[controller]")]
    public class ProductServiceController : Controller
    {
        private IProductRepositoryAsync productRepository;
        public ProductServiceController(IProductRepositoryAsync
                                                productRepository)
        {
            this.productRepository = productRepository;
        }
        [HttpGet]
        public async Task<IActionResult> GetAll()
        {
            var products = await productRepository.SelectAllAsync();
            return Ok(products);
        }
```

Status codes

The Ok method returns an OkObjectResult



Dependency injection

Register the repository in the ConfigureServices method of the Startup class

```
//services registered with AddTransient are disposed after the request
services.AddTransient<IProductRepositoryAsync, ProductRepository>(ctx =>
{
    EcommerceContext context = ctx.GetService<EcommerceContext>();
    return new ProductRepository(context);
});
```

Route parameters

"{id}" is a placeholder variable for the ID of the Product. The id parameter is configured in the MapRoute method in Startup.cs

```
[HttpGet("{id}")]
public async Task<IActionResult> GetByName(string id)
{
    var products = await productRepository.SelectByNameAsync(id);
    if (products == null)
    {
        return NotFound();
    }
    return Ok(products);
}
```

Route constraints

p\\d+ is a regular expression that matches the character p, followed by 1 or more digits. See regxr.com

The Name property is used to generate a link in the Create method

```
[HttpGet("{id:regex(^p\\d+$)}", Name = "GetProductsById")]
public async Task<IActionResult> GetById(string id)
{
    var item = await productRepository.SelectByIdAsync(id);
    if (item == null) {
        return NotFound();
    }
    return Ok(item);
}
```

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-2.1#route-constraint-reference

HttpPost

The FromBody attribute binds the Product parameter's properties to the JSON properties in the HTTP request

```
[HttpPost]
public async Task<IActionResult> Create([FromBody] Product product)
{
    if (product == null )
    {
        return BadRequest(ModelState);
    }

    bool created = await productRepository.CreateAsync(product);
    if (!created)
    {
        return BadRequest($"{product.Id} already exists");
    }

    return CreatedAtRoute("GetProductsById",
                                    new { id = product.Id }, product);
}
```

Web API Client

Add a .NET Standard Class Library project to the solution. Add the Microsoft.AspNet.WebApi.Client assembly from Nuget

```
namespace ClassLibrary.WebApiClient
{
    public class ProductRepository : IProductRepositoryAsync
    {
        private string uri;
        public ProductRepository (string uri) => this.uri = uri;

        public async Task<bool> CreateAsync(Product product)
        {
            using (HttpClient client = new HttpClient())
            {
                //Send the product, encoded as JSON, in a POST request
                                                              to the specified Uri
                HttpResponseMessage response = await client.PostAsJsonAsync(
                                                                    uri, product);
                return response.StatusCode == HttpStatusCode.Created;
            }
        }

        public async Task<ICollection<Product>> SelectAllAsync()
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response = await client.GetAsync(uri);
                //string json = await response.Content.ReadAsStringAsync();
                //HttpStatusCode statusCode = response.StatusCode;
                IEnumerable<Product> products =
                    await response.Content.ReadAsAsync<IEnumerable<Product>>();
                return products.ToList();
            }
        }

        public async Task<Product> SelectByIdAsync(string id)
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response = await client.GetAsync(uri + id);
                return await response.Content.ReadAsAsync<Product>();
            }
        }

        public async Task<ICollection<Product>> SelectByNameAsync(string name)
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response = await client.GetAsync(uri + name);
                IEnumerable<Product> products =
                    await response.Content.ReadAsAsync<IEnumerable<Product>>();
                return products.ToList();
            }
        }
```

```
        public async Task<bool> UpdateAsync(Product product)
        {
            using (HttpClient client = new HttpClient())
            {
                //Send the product, encoded as JSON, in a POST request
                                                to the specified Uri
                HttpResponseMessage response =
                            await client.PutAsJsonAsync(uri+product.Id, product);
                return response.StatusCode == HttpStatusCode.OK;
            }
        }

        public async Task<bool> DeleteAsync(string id)
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response = await client.DeleteAsync(uri + id);
                return response.StatusCode == HttpStatusCode.OK;
            }
        }
    }
}
```

Integration Test

```
namespace ClassLibrary.WebApiClient.Test
{
    [Collection("Collection 1")]
    public class ProductRepositoryIntegrationTest
    {
        private string url = "http://sdineen.uk/api/productservice/";

        [Fact]
        public async void SelectAllAsync_Should_Return_All_Products()
        {
            var productRepository = new ProductRepository(url);
            ICollection<Product> products = await productRepository.SelectAllAsync();
            Assert.Equal(10, products.Count);
        }
    }
}
```

WPF

Introduction

Windows Presentation Foundation is a system for building desktop client applications for Windows. One feature that distinguishes it from the earlier Windows Forms is the separation of presentation from application logic. XAML (an XML-based language) describes the layout of the components on the user interface, while a C# code-behind class handles events. The default XAML namespace brings in CLR namespaces in the PresentationFramework assembly including System.Windows, System.Windows.Controls and System.Windows.Data. The x namespace abbreviation brings in XAML language features, including the Class directive for joining markup and code-behind.

Add a WPF project to the solution and drag the TextBox, Button and TextBlock components from the toolbox.

```
<Window x:Class="WinClient.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WinClient"
        mc:Ignorable="d"
        Title="MainWindow" Height="300" Width="300">
    <Grid>
        <TextBox x:Name="textBox" HorizontalAlignment="Left" Height="23"
              Margin="26,24,0,0" TextWrapping="Wrap" Text="1000000"
              VerticalAlignment="Top" Width="120"/>
        <Button x:Name="button" Content="Start" HorizontalAlignment="Left"
              Margin="26,68,0,0" VerticalAlignment="Top" Width="75"/>
        <Label x:Name="label" Content="Label" HorizontalAlignment="Left"
              Margin="26,112,0,0" VerticalAlignment="Top"/>
    </Grid>
</Window>
```

Events

Events enable an object to notify other objects when something of interest occurs.

To respond to the button being clicked, add the following to the constructor in the code behind class:

```
button.Click += TAB
```

This will generate a method that will be called when the event is raised.

Click is an Event field in the Button class, which would be written in C# as follows:

```
public event RoutedEventHandler Click;
```

Events are invoked from within the class that declares them:

```
Click(this, e)
```

Delegates

Delegates enable methods to be passed as arguments to other methods. A delegate represents a method with a particular parameter list and return type. For example, RoutedEventHandler is a delegate associated with a method that has a void return type and takes two arguments of type Object and RoutedEventArgs. The following method matches the signature of the delegate:

```
private void Button_Click (object sender, RoutedEventArgs e)
{
}
```

Like classes, delegates can be instantiated. The argument to a delegate's constructor is a method name. For example:

```
RoutedEventHandler buttonHandler = new RoutedEventHandler(Button_Click);
```

This delegate instance can be associated with the Click event of the button

```
button.Click += buttonHandler;
```

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    int limit = Convert.ToInt32(textBox.Text);
    int result = (from n in Enumerable.Range(2, limit)
                  where Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)
                  select n).Count();
    sw.Stop();
    textBlock.Text = $"{result} prime numbers. Calculated in
                                    {Math.Round(sw.Elapsed.TotalSeconds, 2)} seconds";
}
```

Improvements

- Layout that adapts to different screen sizes
- Centrally defined styles defining component properties
- Asynchronous methods

Layouts

WrapPanel

```
<WrapPanel Orientation="Vertical">
    <TextBox Name="textBox" Width="100"/>
    <Button Name="button" Content="Start" Margin="0,5,0,5"/>
    <Label Name="label" Content="Label" />
</WrapPanel>
```

StackPanel

```
<StackPanel Orientation="Vertical">
    <TextBox Name="textBox" Width="100" HorizontalAlignment="Left"/>
    <Button Name="button" Content="Start" Width="100" HorizontalAlignment="Left"
                                                          Margin="0,5,0,5"/>
    <Label Name="label" Content="Label" Width="100" HorizontalAlignment="Left" />
</StackPanel>
```

DockPanel

```
<DockPanel>
    <TextBox DockPanel.Dock="Top" Name="textBox" Width="100"
                                                    HorizontalAlignment="Left"/>
    <Label DockPanel.Dock="Bottom" Name="label" Content="Label" Width="100"
                                                    HorizontalAlignment="Left"/>
    <Button DockPanel.Dock="Left" Name="button" Content="Start" Margin="0,5,0,5"/>
    <Rectangle/>
</DockPanel>
```

Grid

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <TextBox Grid.Row="0" Name="textBox" Width="100" Height="30"
                                                    HorizontalAlignment="Left" />
    <Button Grid.Row="1" Name="button" Content="Start"
                            Width="100" Height="30" HorizontalAlignment="Left" />
    <Label Grid.Row="2" Name="label" Content="Label" Width="100" Height="30"
                                                    HorizontalAlignment="Left" />
</Grid>
```

Styles

Styles facilitate applying standard formatting to a set of controls. They can be defined for a window or the whole application.

```
<Window.Resources>
    <Style TargetType="Button">
        <Setter Property="HorizontalAlignment" Value="Left" />
        <Setter Property="Width" Value="100" />
    </Style>
</Window.Resources>
```

Asynchronous and concurrent methods

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        button.Click += Button_Click;
    }

    private async void Button_Click(object sender, RoutedEventArgs e)
    {
        Stopwatch sw = new Stopwatch();
        sw.Start();
        int result = await CalculatePrimesAsync(limit);
        sw.Stop();
        label.Content = $"{result} prime numbers. Calculated in {
                        Math.Round(sw.Elapsed.TotalSeconds, 2)} seconds";
    }

    public async Task<int> CalculatePrimesAsync(int limit)
    {
        Func<int> func = () => (
           from n in Enumerable.Range(2, limit).AsParallel()
           where Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)
```

```
            select n).Count();
        int result = await Task.Run(func); //returns Task<int>
        return result;
    }
```

Task.Run queues the specified work to run on the thread pool and returns a proxy for the task returned by function.

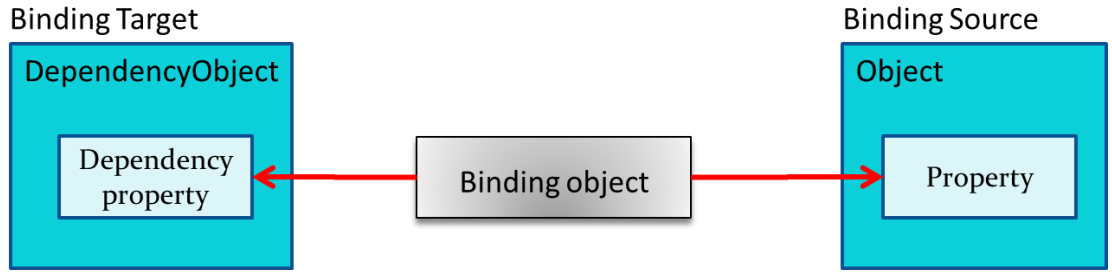
The await operator is applied to a task in an asynchronous method to suspend the execution of the method until the awaited task completes. Methods using await must be modified by the async keyword.

Products

Add a new window with a Label and a TextBox. Double click the TextBox to generate the event handler.

```
<Window x:Class="WpfApplication1.ProductList"
    <DockPanel>
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
            <Label Content="Search" />
            <TextBox Name="SearchBox" Width="100"
                     TextChanged="SearchBox_TextChanged" />
        </StackPanel>
        <DataGrid>
        </DataGrid>
    </DockPanel>
</Window>
```

Data Binding




```
{Binding Path=PropertyName}
```

A binding object connects the properties of binding target object to a data source (for example, a database, an XML file, or any object that contains data). The above XAML markup extension syntax can be used to build a Binding object and set its Path property.

```
        <DataGrid Grid.Row="1" AutoGenerateColumns="False"
                  GridLinesVisibility="None" ItemsSource="{Binding}" >
            <DataGrid.Columns>
                <DataGridTextColumn Binding="{Binding Path=Name}" Header="Name"/>
            </DataGrid.Columns>
        </DataGrid>
```

Set the DataGrid's DataContext in the code behind class

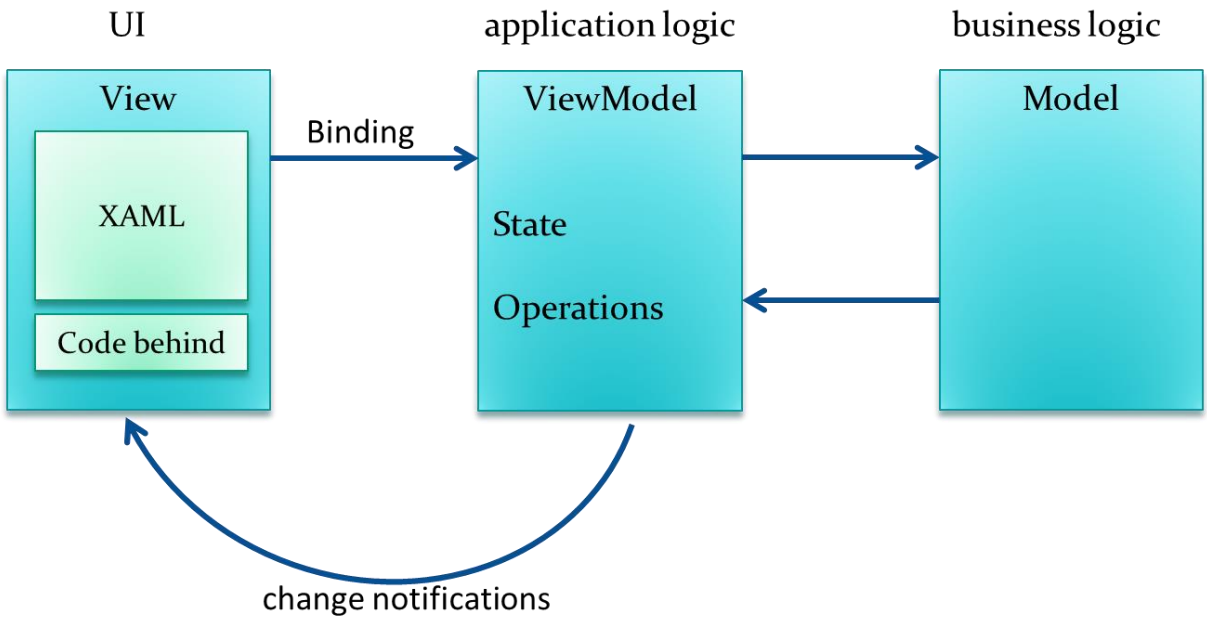
```
namespace WinClient
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private IProductRepositoryAsync productRepository = new ProductRepository(
                                                "http://sdineen.uk/api/productservice/");
        public MainWindow()
        {
            InitializeComponent();
            SearchBox.Focus();
            SearchBox_TextChanged(null, null);
        }

        private async void SearchBox_TextChanged(object sender,
                                                                TextChangedEventArgs e)
        {
            ICollection<Product> products =
                            await productRepository.SelectByNameAsync(SearchBox.Text);
            DataContext = products;
        }
    }
}
```

WPF MVVM

Products

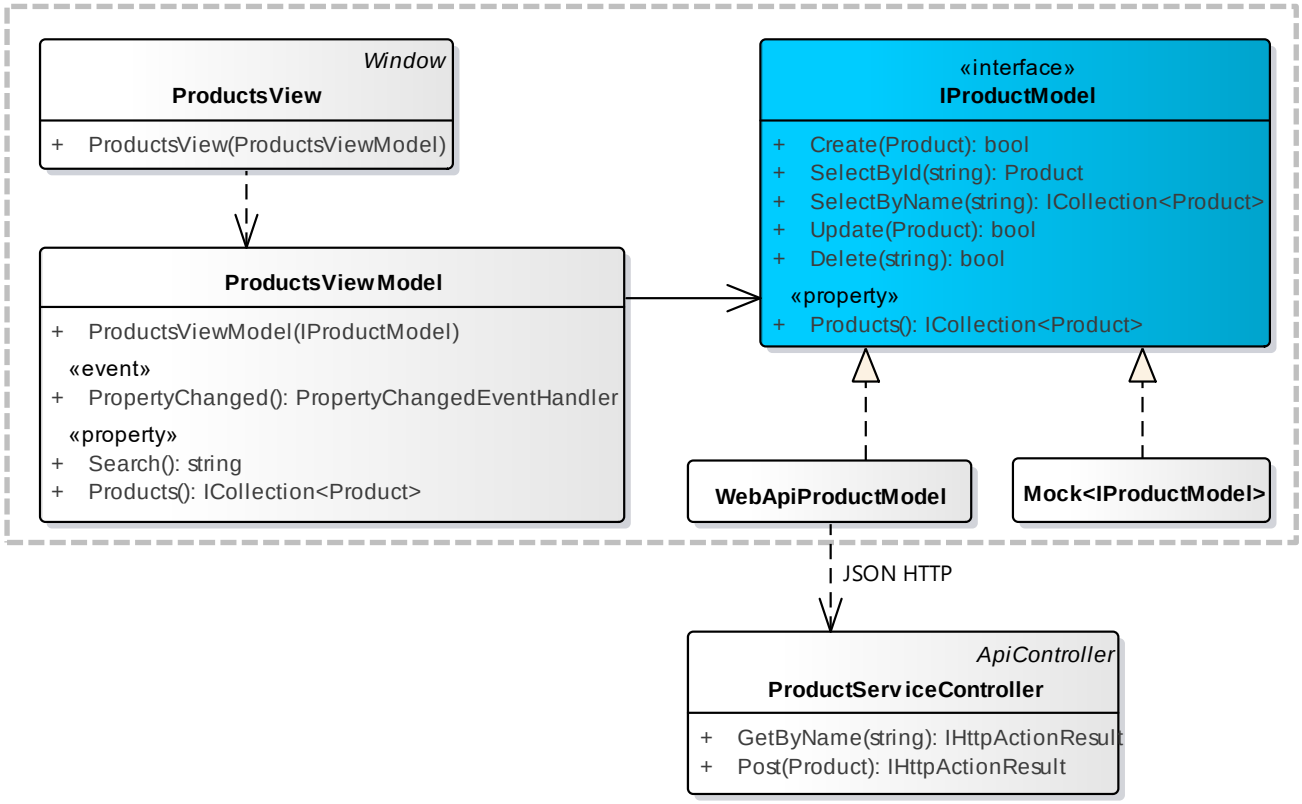
overview

The Model-View-ViewModel (MVVM) pattern facilitates the separation of an application's business and presentation logic from its user interface, making the application easier to test and maintain.



The ViewModel is an abstraction of the View. UI components are bound to properties in the ViewModel. By implementing the INotifyPropertyChanged interface, the ViewModel notifies to the View about changes to its properties. The ViewModel can also contain operations, handling component events.

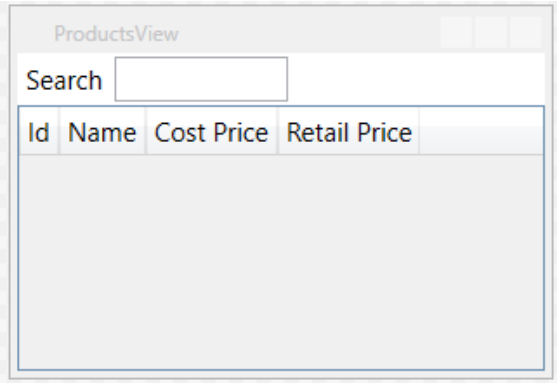
UML



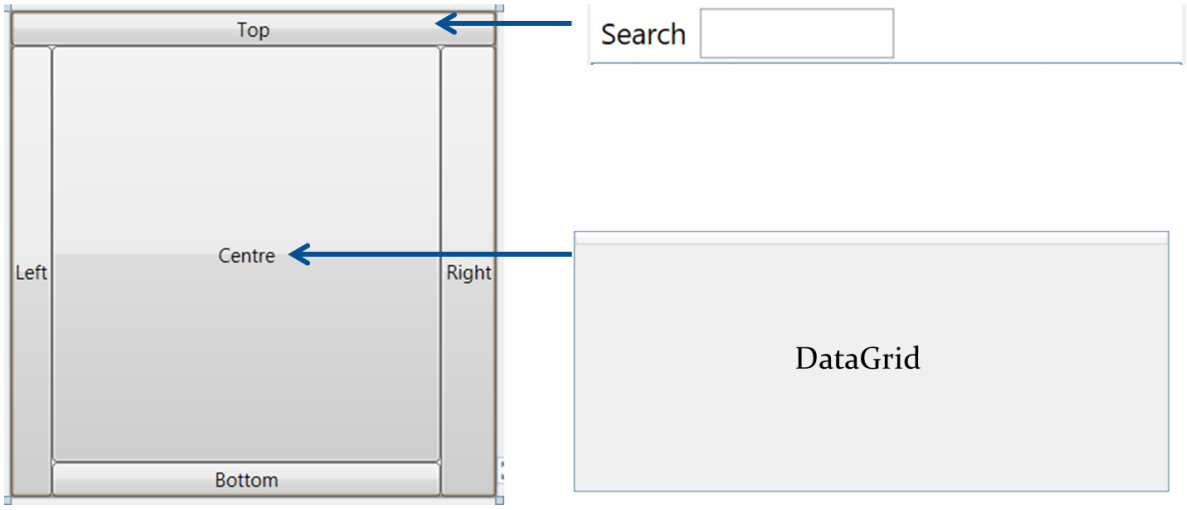
View

Components can be arranged in the window using layout controls, which can be nested.

Canvas	position defined relative to top left of canvas
DockPanel	align content to top, bottom, left right
Grid	arranges content within a grid
StackPanel	content placed in a single row or column
WrapPanel	wraps content to next row or column
TabControl	multiple items share same space on the screen



For example, to build this UI, a DataGrid could be placed in the centre region of a DockPanel, and a StackPanel containing TextBlock and TextBox components placed in the north region.




```
<Window x:Class="WinClient.Products.View.ProductsView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <DockPanel>
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
            <Label Content="Search" />
            <TextBox Name="SearchBox" />
        </StackPanel>
        <DataGrid >
        </DataGrid>
    </DockPanel>
</Window>
```

Binding

The DataContext is the default source for bindings. The code behind class for the window sets the DataContext as the ViewModel.

The TextBox is bound to the ViewModel's Search property. The Mode and UpdateSourceTrigger properties specify that the TextBox will set the ViewModel property as text is typed into it.

The {Binding} markup extension creates a Binding object. Path, the path to the binding source, is a default property, so {Binding Search} is equivalent to {Binding Path=Search}.

```
<StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
    <Label Content="Search" />
    <TextBox Name="SearchBox" Text="{Binding Search, Mode=TwoWay,
                                     UpdateSourceTrigger=PropertyChanged}" />
</StackPanel>
```

The DataGrid's ItemsSource property is bound to the ViewModel's Products property, which could be a collection of Product objects. This contains DataGridTextColumns, which are bound to properties of the Product.

```
<DataGrid ItemsSource="{Binding Products}"
    GridLinesVisibility="None"
```

```
    AutoGenerateColumns="False" >
    <DataGrid.Columns>
       <!-- Binding Property sets the binding that associates
       the column with a property in the data source   -->
       <DataGridTextColumn Binding="{Binding Id}" Header="Id"/>
       <DataGridTextColumn Binding="{Binding Name}" Header="Name"/>
       <DataGridTextColumn Binding="{Binding CostPrice}" Header="Cost Price"/>
       <DataGridTextColumn Binding="{Binding RetailPrice}" Header="Retail Price"/>
    </DataGrid.Columns>
</DataGrid>
```

View code behind

This sets the DataContext for the window as the ViewModel

```
    public partial class ProductsView : Window
    {
        public ProductsView(ProductsViewModel viewModel)
        {
            InitializeComponent();
            DataContext = viewModel;
            SearchBox.Focus();
        }
    }
```

ViewModel

By implementing INotifyPropertyChanged, subscribers can be notified when the properties change

```
namespace WinClient.Products.ViewModel
{
    public class ProductsViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        private IProductRepositoryAsync productRepository;

        private string search;
        private ICollection<Product> products;
        public ProductsViewModel(IProductRepositoryAsync productRepository)
        {
            this.productRepository = productRepository;
            LoadData();
        }

        private async void LoadData()
        {
            Products = await productRepository.SelectByNameAsync(Search);
        }

        public string Search
        {
            get { return search; }
            set
            {
                search = value;
                LoadData();
            }
        }
```

```
        public ICollection<Product> Products  //Bound to DataGrid ItemsSource
        {
            get { return products; }
            set
            {
                products = value;
                //PropertyChanged null if there are no subscribers to the event,
                //due to DataContext not being set
                PropertyChanged?.Invoke(this,
                                        new PropertyChangedEventArgs("Products"));
            }
        }
    }
}
```

Dependency injection

Unity is a lightweight, extensible dependency injection container that supports interception, constructor injection, property injection, and method call injection. Other popular containers include Ninject and Autofac.

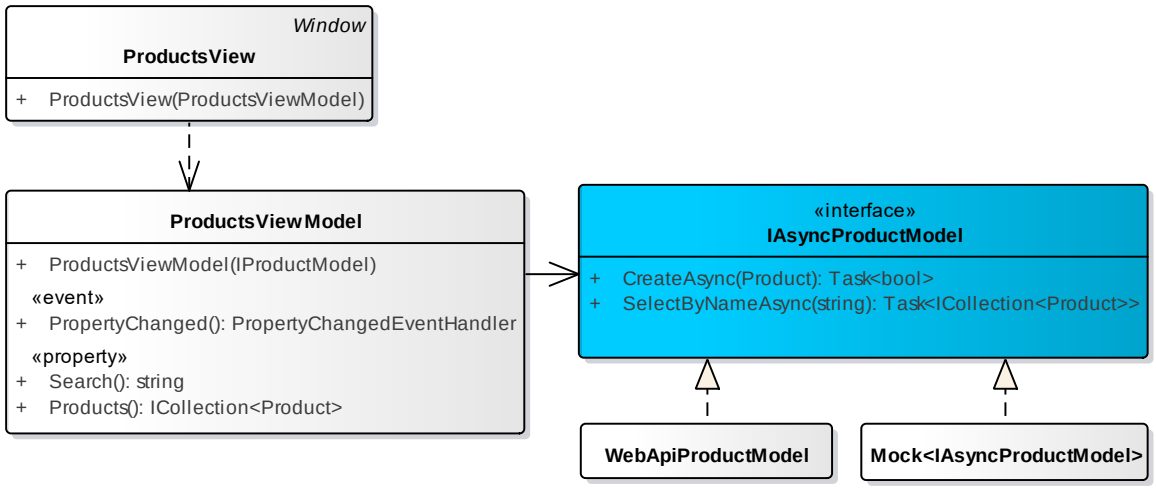
Unity's RegisterType method associates an implementing class with an interface. The Resolve method uses constructor injection to instantiate the dependencies.

Instead of the StartupUri property in App.xaml, add an OnStartup event handler to App.xaml.cs

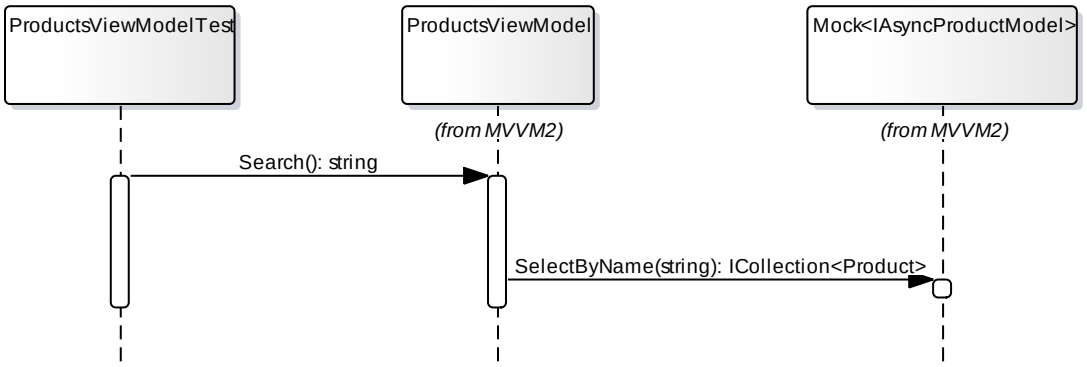
The ProductsViewModel constructor takes an interface argument, so need to register association between IProductRepositoryAsync and implementing class. ProductRepository constructor takes string argument, which is passed in using InjectionConstructor

```
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            string url = "http://sdineen.uk/api/productservice/";
            IUnityContainer container = new UnityContainer();
            container.RegisterType<IProductRepositoryAsync, ProductRepository>(
                                                    new InjectionConstructor(url));
            ProductsView view = container.Resolve<ProductsView>();
            view.Show();
        }
    }
```

Unit tests



Verify that the DOC's SelectByName method is called when the SUT's Search property is set.




```
namespace WinClient.Tests.ViewModel
{
    public class ProductsViewModelTest
    {
        private Mock<IAsyncProductModel> doc = new Mock<IAsyncProductModel>();
        private ProductsViewModel sut;

        public ProductsViewModelTest()
        {
            sut = new ProductsViewModel(doc.Object);
        }

        [Fact]
        [Trait("WinClient", "Unit Test")]
        public void SearchPropertySetter_ShouldCallSelectByNameAsync()
        {
            // Act
            sut.Search = "something";
            // Assert
            doc.Verify(m => m.SelectByNameAsync("something"));
        }
```

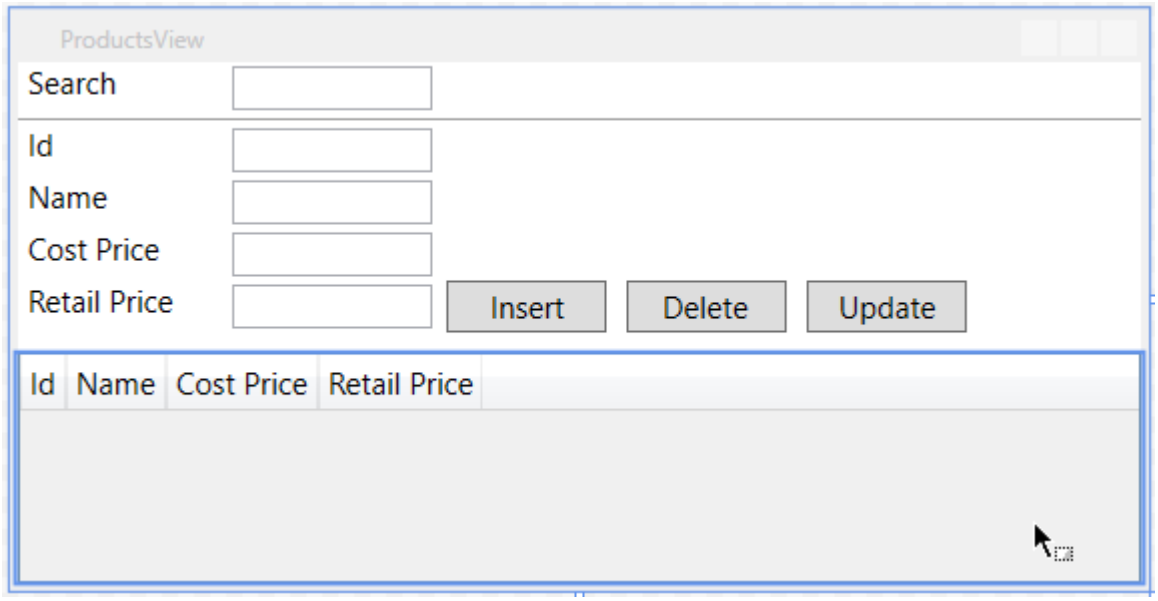
Extract the IAsyncProductModel interface from the WebApiProductModel class. Change the ProductsViewModel constructor to take an interface argument. If using Unity DI, associate this interface with a class in App.xaml.cs

```
container.RegisterType<IAsyncProductModel, WebApiProductModel>();
```

Products MVVM continued

UI

The application could be embellished by enabling inserts, updates and deletes to the database via the web service



View

The additional TextBoxes and Buttons are bound to properties in the ViewModel.

```
            <Separator/>
            <StackPanel Orientation="Horizontal">
                <Label Content="Id" />
                <TextBox Text="{Binding Id}" />
            </StackPanel>
            <StackPanel Orientation="Horizontal">
                <Label Content="Name" />
                <TextBox Text="{Binding Name}" />
            </StackPanel>
            <StackPanel Orientation="Horizontal">
                <Label Content="Cost Price" />
                <TextBox Text="{Binding CostPrice}" />
            </StackPanel>
            <StackPanel Orientation="Horizontal" Margin="0,0,0,10">
                <Label Content="Retail Price" />
                <TextBox Text="{Binding RetailPrice}" />
                <Button Command="{Binding AddProductCommand}" Content="Insert" />
                <Button Command="{Binding DeleteProductCommand}" Content="Delete" />
                <Button Command="{Binding UpdateProductCommand}" Content="Update" />
            </StackPanel>
```

Styles

Much like CSS styles for an HTML page, the properties of XAML controls can be configured such that all the components of a specified type are rendered in the same way. The following style elements set the FontSize, MinWidth and Margin properties for all the Label, TextBox, DataGridCell and DataGridColumnHeader controls in the Window.

```
<Window x:Class="WinClient.Products.View.ProductsView1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Window.Resources>
        <Style TargetType="Label">
            <Setter Property="FontSize" Value="15" />
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="FontSize" Value="15" />
            <Setter Property="MinWidth" Value="100" />
            <Setter Property="Margin" Value="2" />
        </Style>
        <Style TargetType="DataGridCell">
            <Setter Property="FontSize" Value="15" />
        </Style>
        <Style TargetType="DataGridColumnHeader">
            <Setter Property="FontSize" Value="15" />
        </Style>
    </Window.Resources>
```

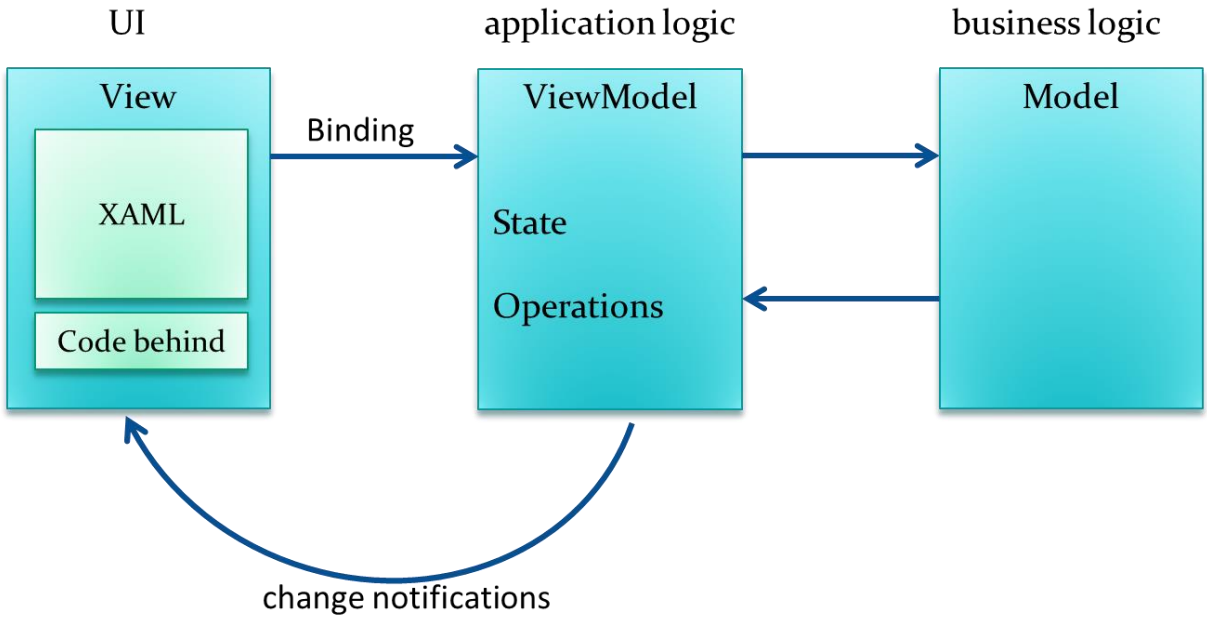
ViewModel

```
namespace WinClient.Products.ViewModel
{
    public class ProductsViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        private IAsyncProductModel productModel;

        //Backing fields for properties bound to TextBoxes
        private string id;
        private string name;
        private double? costPrice;
        private double? retailPrice;
        private string search;
        //Backing fields for properties bound to DataGrid
        private ICollection<Product> products;
        private Product selectedProduct;

        public ProductsViewModel(IAsyncProductModel productModel)
        {
            this.productModel = productModel;
            LoadCommands();
            LoadData();
        }

        private void LoadCommands()
        {
            AddProductCommand = new CustomCommand(AddProduct, CanAddProduct);
            DeleteProductCommand = new CustomCommand(DeleteProduct, CanDeleteProduct);
            UpdateProductCommand = new CustomCommand(UpdateProduct, CanUpdateProduct);
        }

        private async void LoadData()
        {
            Products = await productModel.SelectByNameAsync(Search);
        }

        #region Bound Properties
        public string Search
        {
            get { return search; }
            set
            {
                search = value;
                LoadData();
            }
        }
        public Product SelectedProduct //Bound to DataGrid SelectedItem
        {
            get { return selectedProduct; }
            set
            {
                selectedProduct = value;
            }
        }



        //Operations
        public CustomCommand AddProductCommand { get; private set; }
```

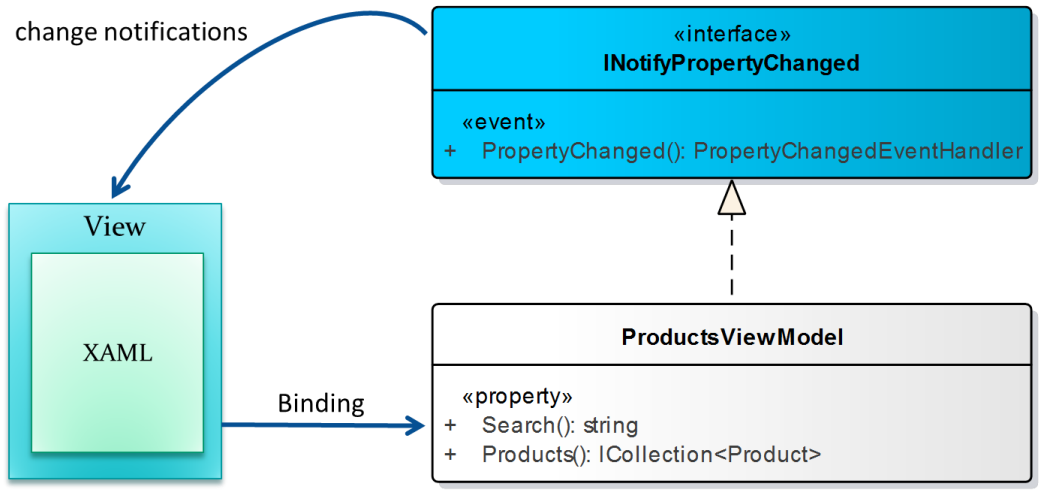
```
public CustomCommand DeleteProductCommand { get; private set; }
public CustomCommand UpdateProductCommand { get; private set; }

private async void AddProduct(object obj)
{
    Product product = new Product(Id, Name, CostPrice.GetValueOrDefault(),
                                          RetailPrice.GetValueOrDefault());
    await productModel.CreateAsync(product);
    //continues when awaitable Task returns
    LoadData();
}

private bool CanAddProduct(object obj)
{
    //determines whether button is enabled
    return Id != null && Name != null && CostPrice.HasValue
                                          && RetailPrice.HasValue;
}

private async void UpdateProduct(object obj)
{
    //Pass the SelectedProduct to the IProductModel's
    //UpdateAsync method
}


//the other properties bound to components in the View aren't shown
```

Primes MVVM

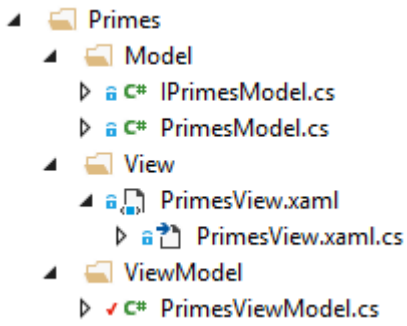
The Model-View-ViewModel (MVVM) pattern facilitates the separation of an application's business and presentation logic from its user interface, making the application easier to test and maintain.



The ViewModel is an abstraction of the View. UI components are bound to properties in the ViewModel. By implementing the INotifyPropertyChanged interface, the ViewModel notifies to the View about changes to its properties. The ViewModel can also contain operations, handling component events.



Folders



Model

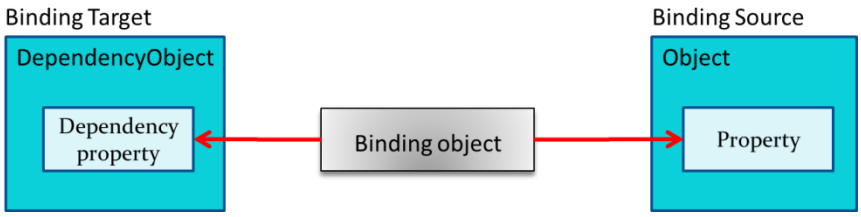
```
namespace WinClient.MVVM.Model.Primes
{
    public class PrimesModel
    {
        public int Count(int max)
        {
            return (from n in Enumerable.Range(2, max)
                    where Enumerable.Range(2, (int)Math.Sqrt(n)-1).All(i => n % i > 0)
                    select n).Count();
        }

        public async Task<int> CountAsync(int max)
        {
            Func<int> func = () =>
                 (from n in Enumerable.Range(2, max)//.AsParallel()
                  where Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)
                  select n).Count();
            int result = await Task.Run(func);
            return result;
        }
    }
}
```

Task.Run queues the specified work to run on the thread pool and returns a proxy for the task returned by function.

The await operator is applied to a task in an asynchronous method to suspend the execution of the method until the awaited task completes. Methods using await must be modified by the async keyword.

Extract the interface for the above class.

View



A binding object connects the properties of binding target object to a data source (for example, a database, an XML file, or any object that contains data). The XAML markup extension syntax enclosed in braces:

```
<TextBlock Text="{Binding Result}"
```

is used to build a Binding object. The default source of bindings is the DataContext property, which can be set in the code-behind class. The constructor takes a PrimesViewModel argument, which can be generated.

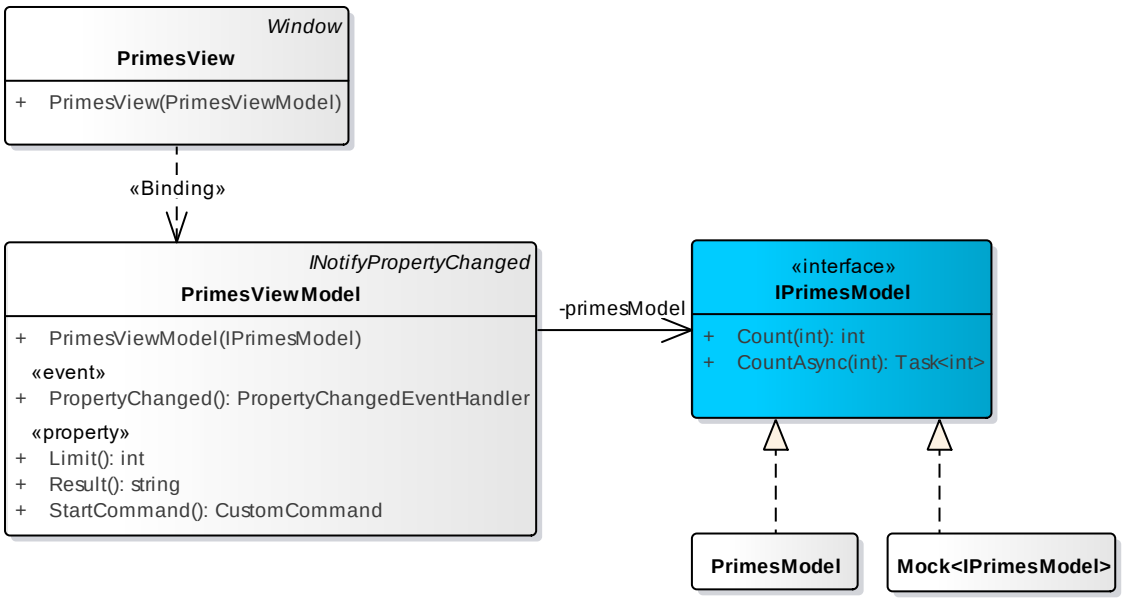
```
namespace WinClient.MVVM.View
{
    public partial class PrimesView : Window
    {
        public PrimesView(PrimesViewModel viewModel)
        {
            InitializeComponent();
            DataContext = viewModel;
        }
    }
}
```

The TextBox.Text property has a default UpdateSourceTrigger value of LostFocus. Changing this to PropertyChanged will cause the source to be updated as text is typed into the TextBox.

```
    <Grid>
        <TextBox Text="{Binding Limit, UpdateSourceTrigger=PropertyChanged}"
        <Button Command="{Binding StartCommand}"
        <TextBlock Text="{Binding Result}"
    </Grid>
```

ViewModel

The View binds to properties in the ViewModel named Limit, Result and StartCommand


Window
PrimesView
+ PrimesView(PrimesViewModel)
«Binding»
INotifyPropertyChanged
PrimesViewModel
+ PrimesViewModel(IPrimesModel)
«event»
+ PropertyChanged(): PropertyChangedEventHandler
«property»
+ Limit(): int
+ Result(): string
+ StartCommand(): CustomCommand
-primesModel
«interface»
IPrimesModel
+ Count(int): int
+ CountAsync(int): Task<int>
PrimesModel
Mock<IPrimesModel>

```
namespace WinClient.MVVM.ViewModel
{
    public class PrimesViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        private IPrimesModel primesModel;

        private int limit;
        private string result;

        public PrimesViewModel(IPrimesModel primesModel)
        {
            this.primesModel = primesModel;
        }

        public int Limit
        {
            get { return limit; }
            set
            {
                limit = value;
                LoadData();
            }
        }

        public string Result
        {
            get { return result; }
            set
            {
                result = value;
                // Notify listeners <TextBlock Text="{Binding Result}"
                PropertyChanged(this, new PropertyChangedEventArgs("Result"));
            }
        }

        private async void LoadData()
        {
            Stopwatch sw = new Stopwatch();
            sw.Start();
            int count = await primesModel.CountAsync(Limit);
            sw.Stop();
            Result = $"{result} prime numbers. Calculated in
                            {Math.Round(sw.Elapsed.TotalSeconds, 2)} seconds";
        }
    }
}
```

Startup

Since dependencies need to be passed into the View and ViewModel's constructors, the View is instantiated in the App class. Take out the StartupUri property in App.xaml.

```
namespace WinClient
{
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);

            PrimesModel model = new PrimesModel();
            PrimesViewModel viewModel = new PrimesViewModel(model);
            PrimesView view = new PrimesView(viewModel);
            view.Show();
        }
    }
}
```

Dependency injection

Dependency injection is a software design pattern that implements inversion of control for resolving dependencies. An injection is the passing of a dependency to a dependent. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

Benefits of DI include

- Maintainability
- Testability (using mock implementations of injected interfaces)
- Flexibility and Extensibility
- Loose Coupling (reducing the number of dependencies between the application's components)

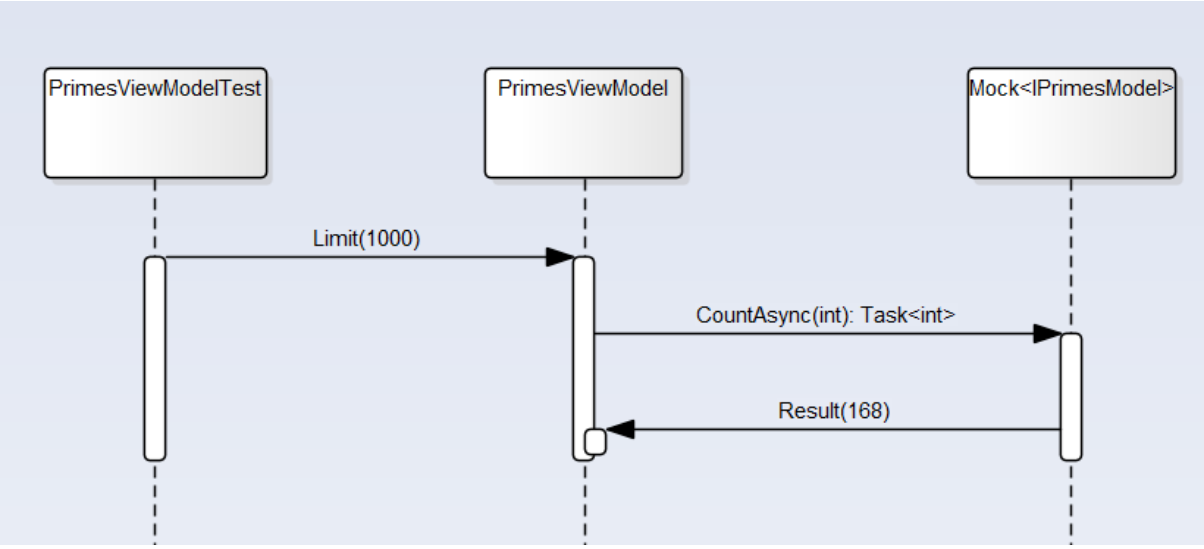
Unity is a lightweight, extensible dependency injection container that supports interception, constructor injection, property injection, and method call injection. Other popular containers include Ninject and Autofac.

Unity's RegisterType method associates an implementing class with an interface. The Resolve method uses constructor injection to instantiate the dependencies.

```
namespace WinClient
{
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);

            IUnityContainer container = new UnityContainer();
            container.RegisterType<IPrimesModel, PrimesModel>();
            container.Resolve<PrimesView>().Show();        }
    }
}
```

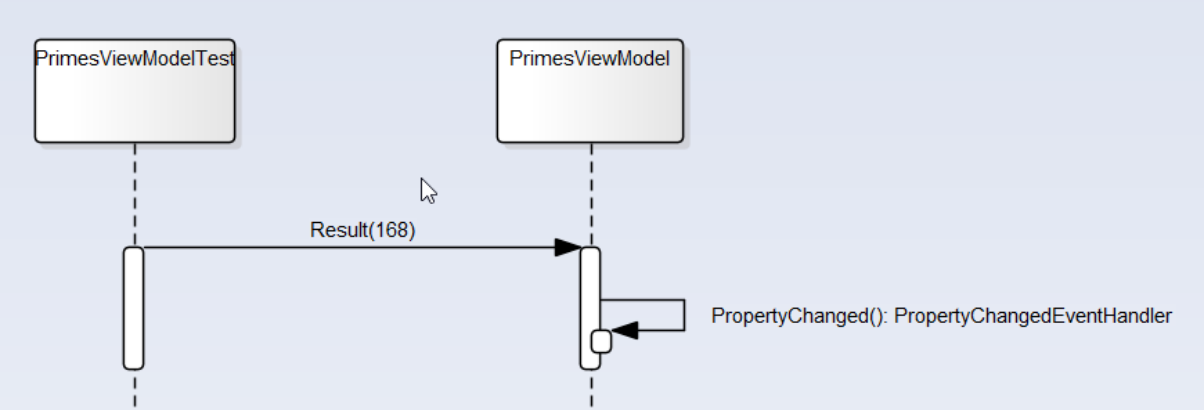
Unit tests

A Mock instance of IPrimesModel can be passed into the PrimeViewModel's constructor, enabling verification of interactions between the system under test and its dependency. For example, when the ViewModel's Limit property is set, the Model's CountAsync method should be called, and the return value assigned to the ViewModel's Result Property.




```
namespace WinClient.Tests
{
    public class PrimesViewModelTest
    {
        private Mock<IPrimesModel> primesModelMock = new Mock<IPrimesModel>();
        private PrimesViewModel viewModel;

        private bool eventRaised;
        private PropertyChangedEventArgs eventArgs;

        //called before each test method
        public PrimesViewModelTest()
        {
            primesModelMock.Setup(pm => pm.CountAsync(1000)).ReturnsAsync(168);
            primesModelMock.Setup(pm => pm.CountAsync(1000000)).ReturnsAsync(78498);
            viewModel = new PrimesViewModel(primesModelMock.Object);

            //add a delegate instance to the ViewModel's PropertyChanged event to
            //enable tracking
            ((INotifyPropertyChanged)viewModel).PropertyChanged +=
                (object sender, PropertyChangedEventArgs e) =>
                                        { eventRaised = true; eventArgs = e; };
        }
```

The following test asserts that when the ViewModel's Result property is set, a PropertyChanged event is raised. The ViewModel implements INotifyPropertyChanged



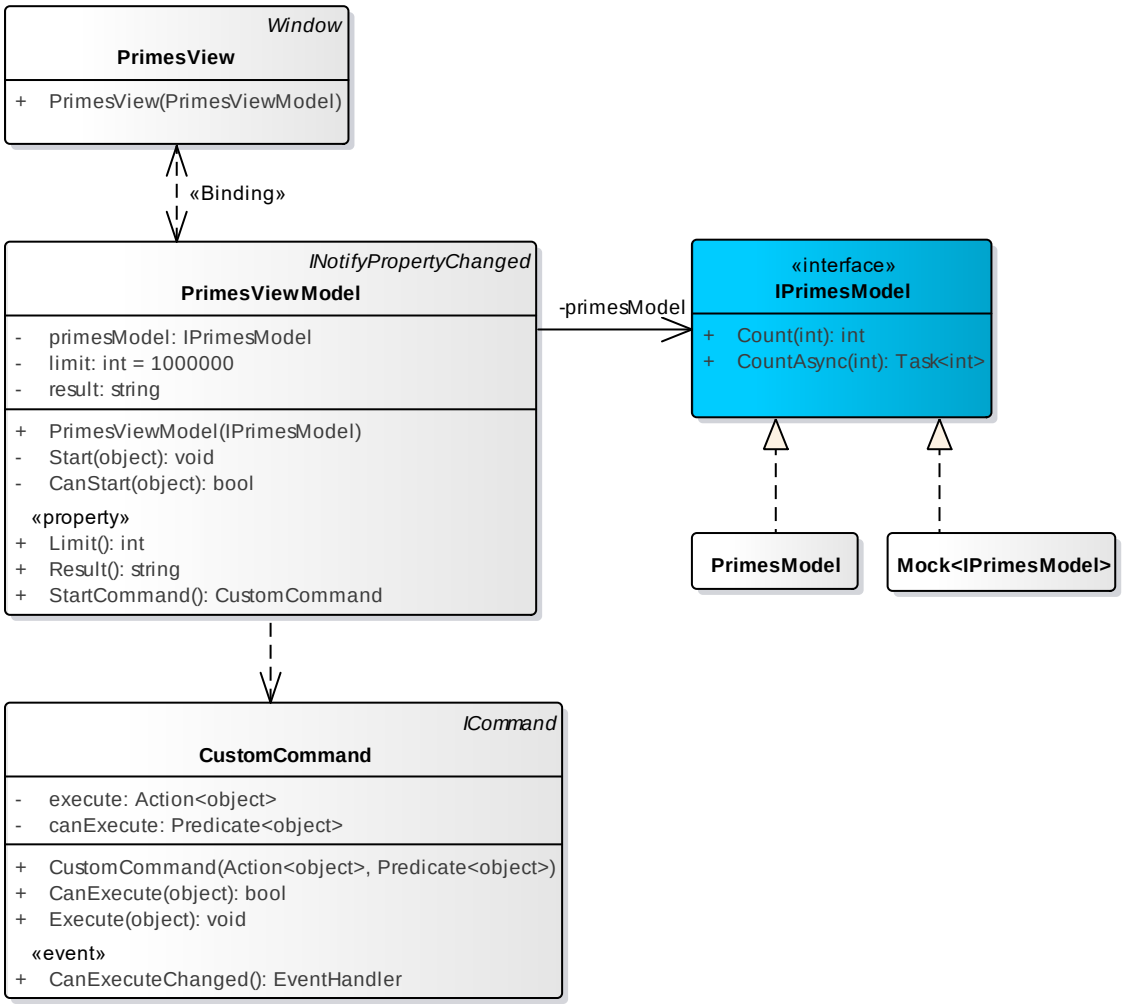

```
[Fact]
[Trait("Category", "Unit Test - Interactions")]
public void ResultProperty_ShouldRaiseEventNamedResult()
{
    // Act
    viewModel.Result = "168";

    // Assert
    Assert.True(eventRaised);
    Assert.Equal("Result", eventArgs.PropertyName);
}
```

Commands

The ViewModel can contain operations as well as state.

```
<Button Command="{Binding StartCommand}"
```

The Command property of the Button is bound to a property that implements ICommand. StartCommand is a property of type CustomCommand. This is a sample implementation of ICommand whose constructor takes two delegate arguments; an Action that determines what the command does and a predicate that determines whether the component is enabled. The Start and CanStart private methods match the signatures of these delegates.




```
namespace WinClient.MVVM.Utility
{
    public class CustomCommand : ICommand
    {
        private Action<object> execute;
        private Predicate<object> canExecute;

        public CustomCommand(Action<object> execute, Predicate<object> canExecute)
        {
            this.execute = execute;
```

```
            this.canExecute = canExecute;
        }

        public bool CanExecute(object parameter)
        {
            return canExecute == null ? true : canExecute(parameter);
        }

        //executes code in ViewModel
        public void Execute(object parameter)
        {
            execute(parameter);
        }

        public event EventHandler CanExecuteChanged
        {
            add
            {
                CommandManager.RequerySuggested += value;
            }
            remove
            {
                CommandManager.RequerySuggested -= value;
            }
        }
    }
}

namespace WinClient.MVVM.ViewModel
{
    public class PrimesViewModel : INotifyPropertyChanged
    {
        private IPrimesModel primesModel;

        public PrimesViewModel(IPrimesModel primesModel)
        {
            this.primesModel = primesModel;
            LoadCommands();
            LoadData();
        }

        public CustomCommand StartCommand { get; private set; }

        private void LoadCommands()
        {
            StartCommand = new CustomCommand(Start, CanStart);
        }

        private void Start(object obj)
        {
            LoadData();
        }

        private bool CanStart(object obj)
        {
            return true;
        }
```