開始前...
了解需求
開始開發
    新增方案
    建立測試專案
    使用 Entity Framework Core 建立 DBContext 及 Model
    建立 Controller
    第一個紅燈
    實作功能
        .NET Core 相依性注入
    NSubstitute
    第一個綠燈
    下一個功能
    最後一個功能

# 開始前...

開始開發之前，你需要先準備好開發環境，以下擇一：

- Visual Studio 2017（若是用較舊版本，則需要再另外安裝 .NET Core 相關套件及工具）

  https://visualstudio.microsoft.com/zh-hant/downloads/?rr=https%3A%2F%2Fwww.google.com.tw%2F
- Visual Studio Code

  https://code.visualstudio.com/download

  > .NET Core 特點之一，不需要再依賴 Visual Studio，且終於可以脫離 Windows 來開發，而 Visual Studio Code 則是微軟推出的一套文字編輯工具，可以想像成 Sublime 或 Notepad++，讓你在 Mac 或是其他 Linux 系統都可以使用

在還不熟悉前，推薦先使用 Visual Studio 來開發

# 了解需求

在任何系統功能開始開發前，為了避免開發出的功能不是使用者所想，或是 over design，我們都必須先了解使用者的需求，就算是工程師也一樣，此次 Demo 專案的需求如下敘述：

*某電商想要有一個商品清單功能，可以**檢視商品清單**，以及**結算使用者選擇商品**的總計金額，此金額要配合某電商當前的折扣優惠活動來計算*

專案採前後端分離，不考慮前端架構，並假設使用者放入購物車的資料皆紀錄在資料庫內，經過規劃後，我們需要提供兩個 API 供前端取用：

- 取得商品清單
- 取得所選商品的總計金額

因為資料記錄在資料庫內，可以先用下列 SQL，在 local database 建立相關的 schema

```
CREATE DATABASE DEMO;
GO
USE DEMO;
GO
ALTER DATABASE DEMO ADD FILEGROUP DEMO_DATA;
GO

ALTER DATABASE DEMO
ADD FILE
(
    NAME = 'DEMO_DATA01',
    -- 依實際想要存放分頁檔的位置
    FILENAME = 'C:\Deploy\Database File\DEMO\DEMO_DATA01.ndf',
    SIZE = 4MB,
    FILEGROWTH = 1MB
)
TO FILEGROUP DEMO_DATA;
GO

CREATE LOGIN prodmgr
    WITH PASSWORD = 'prodmgr',
    DEFAULT_DATABASE = DEMO,
    CHECK_EXPIRATION = OFF,
    CHECK_POLICY = OFF;
GO

CREATE SCHEMA prod;
GO
CREATE USER prodmgr FOR LOGIN prodmgr WITH DEFAULT_SCHEMA = prod;
GO

EXEC sp_addrolemember 'db_owner', 'prodmgr';
```

```
CREATE TABLE prod.Product (
    id              INT IDENTITY    NOT NULL,
    name            NVARCHAR(20)        NOT NULL,
    price           DECIMAL(7, 2)       NOT NULL,
    in_stock        INT                 NOT NULL,
    brief           NVARCHAR(100),
    CONSTRAINT PK_Product PRIMARY KEY (id)
) ON DEMO_DATA;
GO
```

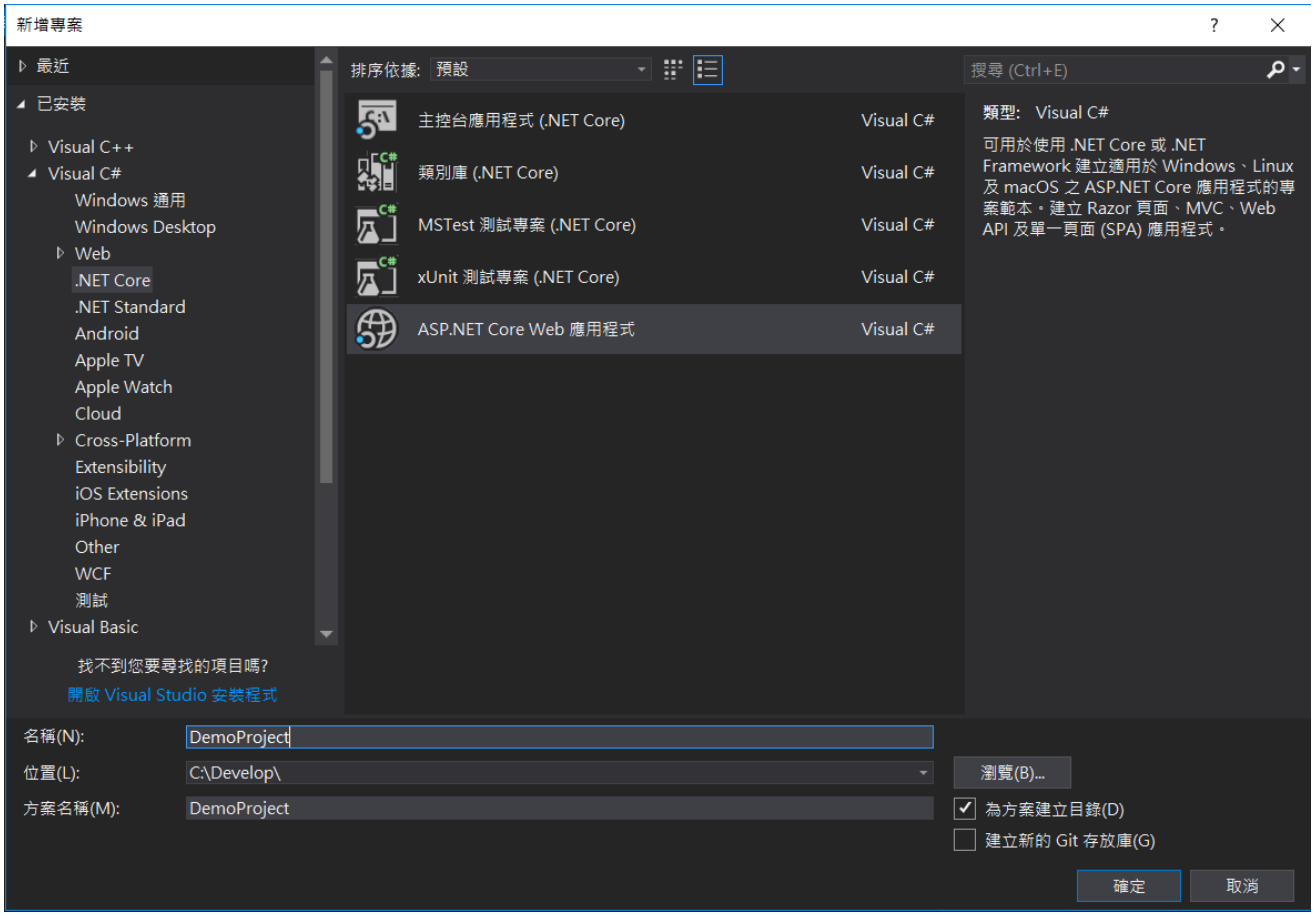
```
INSERT INTO prod.Product (name, price, in_stock, brief)
VALUES ('科幻四小俠', 299, 6, '垃圾書刊'),
       ('宇宙科幻', 560, 2, '暢銷書籍'),
       ('名偵探-ㄎㄅ', 250, 5, '都在唬爛'),
       ('這不是我認識的偵探', 520, 0, '._.');
GO
```

此專案架構及 **table schema** 僅做 **demo** 使用，請勿使用於實務上

# 開始開發

## 新增方案

開啟 VS 2017 並新增一個 ASP.NET Core Web 應用程式

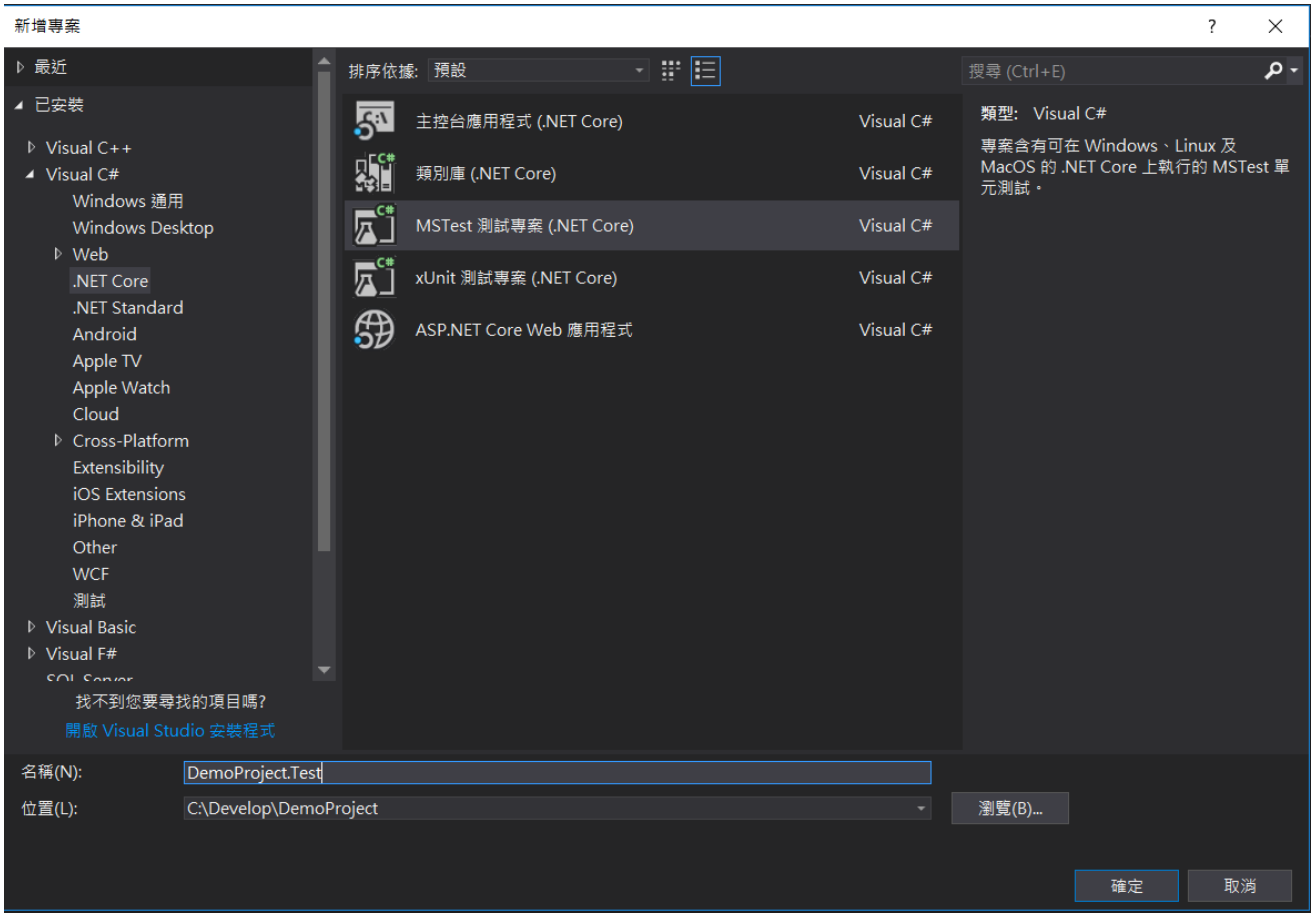



選擇 API 專案


如此已經完成開發 .NET Core 的基本作業，接著建立測試專案

## 建立測試專案

建立測試專案，選擇 MSTest 測試專案 (.NET Core)




針對此次功能，建立一個「商品」測試類別，其中需要測試兩個主要功能及其下的防呆功能：

- 取得商品清單
- 取得所選商品總計金額
    - 取得所選商品總計金額_未選擇商品

測試類別如下 (在測試專案下新增一個類別即可)

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DemoProject.Test
{
    [TestClass]
    public class 商品
    {
        [TestMethod]
        public void 取得商品清單()
        {
            // Arrange
            // Act
            // Assert
        }

        [TestMethod]
        public void 取得所選商品總計金額()
        {
            // Arrange
```

```
            // Act
            // Assert
        }

        [TestMethod]
        public void 取得所選商品總計金額_未選擇商品()
        {
            // Arrange
            // Act
            // Assert
        }
    }
}
```

先關注在「取得商品清單」功能，將其他測試方法註解

取得商品清單：

input－無 對象－商品 controller 動作－取得商品清單 output－商品清單

直觀的測試方法應該如下：

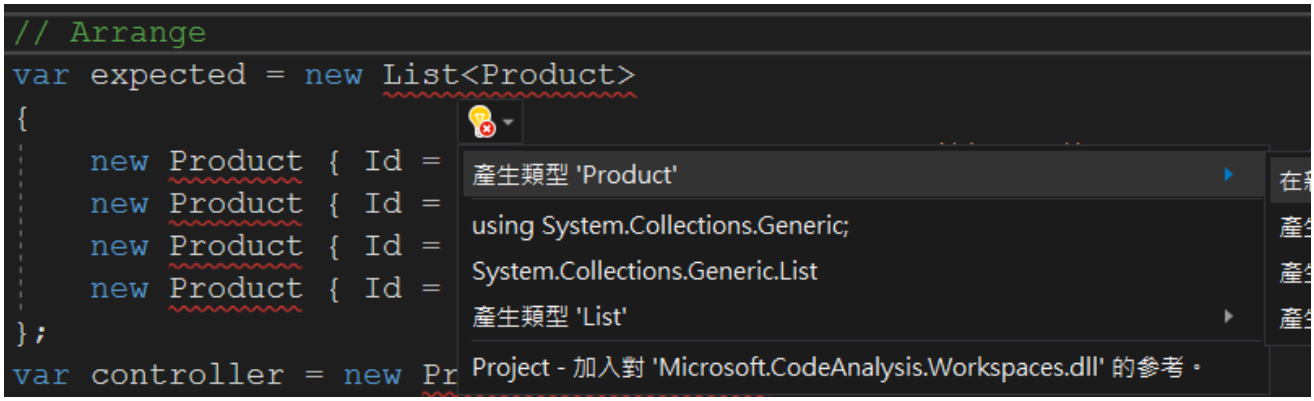
```
[TestMethod]
public void 取得商品清單()
{
    // Arrange
    // 預期結果
    var expected = new List<Product>
    {
        new Product { Id = 1, Name = "科幻四小俠", Price = 299, InStock = 6, Brief = "垃圾書刊"
},
        new Product { Id = 2, Name = "宇宙科幻", Price = 560, InStock = 2, Brief = "暢銷書籍" },
        new Product { Id = 3, Name = "名偵探-ㄎㄅ", Price = 250, InStock = 5, Brief = "都在唬爛"
},
        new Product { Id = 4, Name = "這不是我認識的偵探", Price = 520, InStock = 0, Brief =
"._." }
            };
    // 商品 controller
    var controller = new ProductsController();

    // Act
    var actual = controller.GetAll();

    // Assert
    Assert.AreEqual(expected, actual);
}
```

*直接將上面程式碼貼到測試類別中，可以發現許多錯誤的地方，原因當然是因為我們還沒產生對應的類別及方法，但可以先透過抽象的方式來產生 test code，之後再一一產生對象，如此能幫助我們釐清各方法的職責*

接著讓我們透過 VS 的自動產生來一一建立需要的類別，`ctrl+.` 或是移至錯誤的地方顯示 smart tag，並依據 namespace 來產生類別

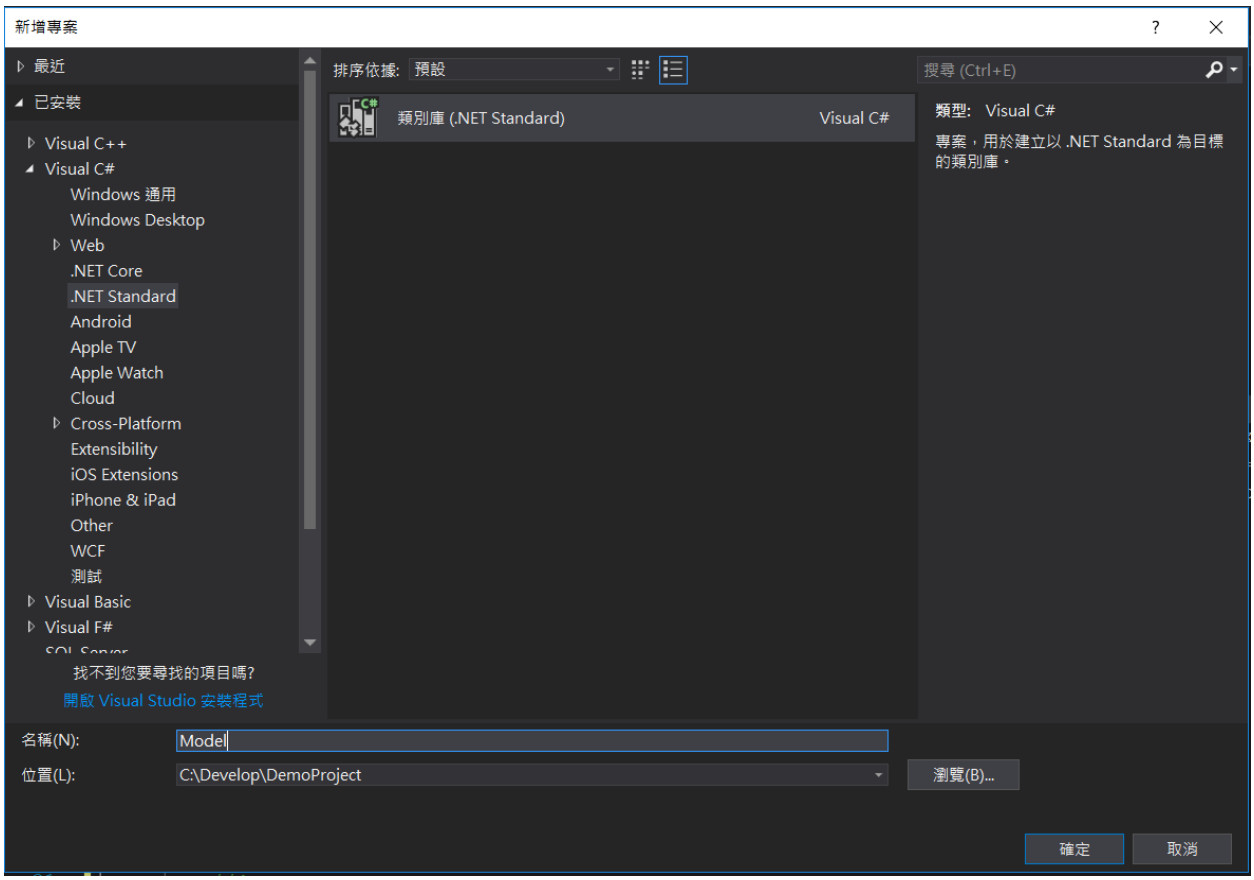



但在 .NET Core API 及 EF Core 框架中，我們有更快產生對應類別的方法，因此我們在這裡以手動的方式來新增這些缺少的類別 (若是之後在類別庫建立類別時，就可以用自動產生的方式來加速開發)

## 使用 Entity Framework Core 建立 DBContext 及 Model

首先建立 Product Model，利用 EF Core 的 database first 來依據 schema 自動建立 Model，另外考量專案架構，我們將 Model 以專案方式獨立出來，之後可以在各層 (邏輯、API 等) 透過參考方式引用

1. 建立 Model 類別庫，注意這裏我們建立的是 .NET Standard 類別庫

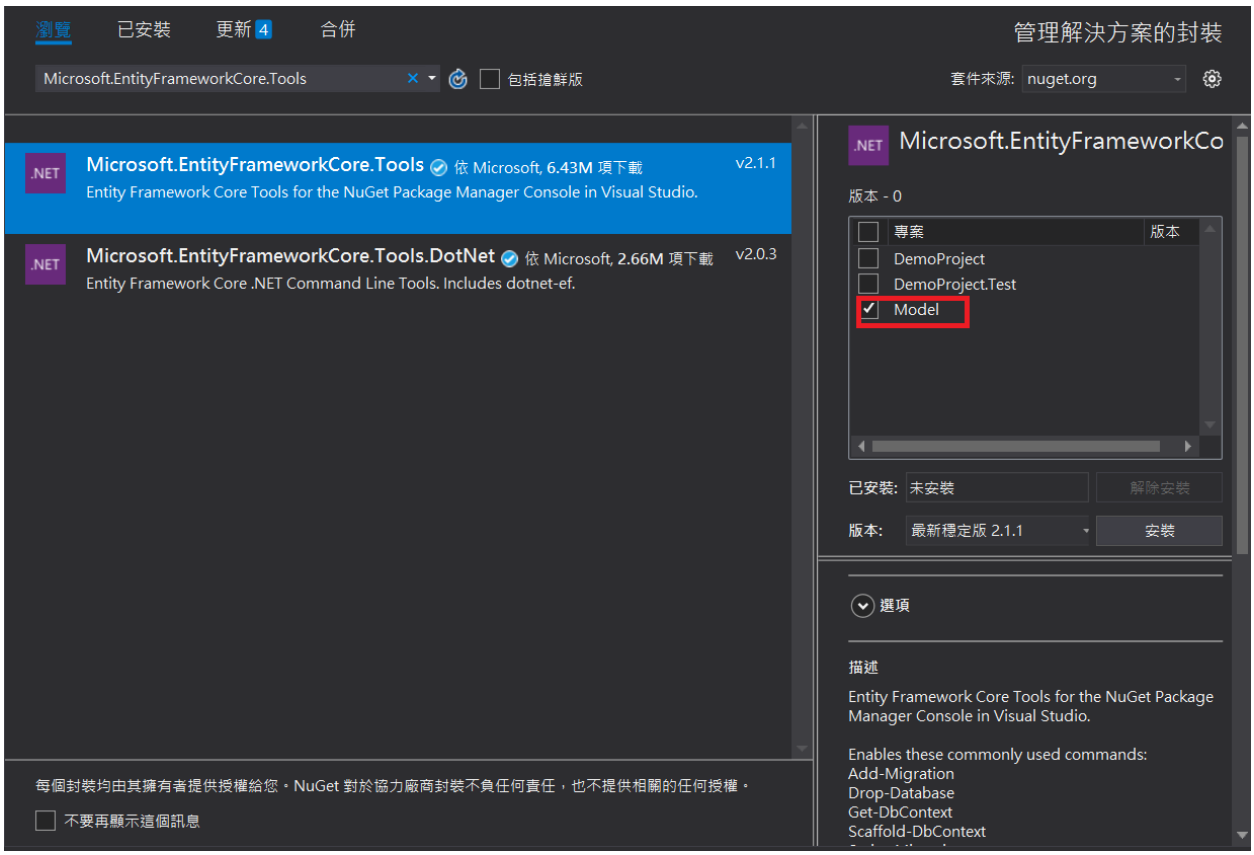



2. 移除預設產生的 Class1.cs
3. 透過 nuget 安裝 EF Core 的相關套件，可以透過 nuget 套件管理主控台，或是透過 nuget 管理工具安裝

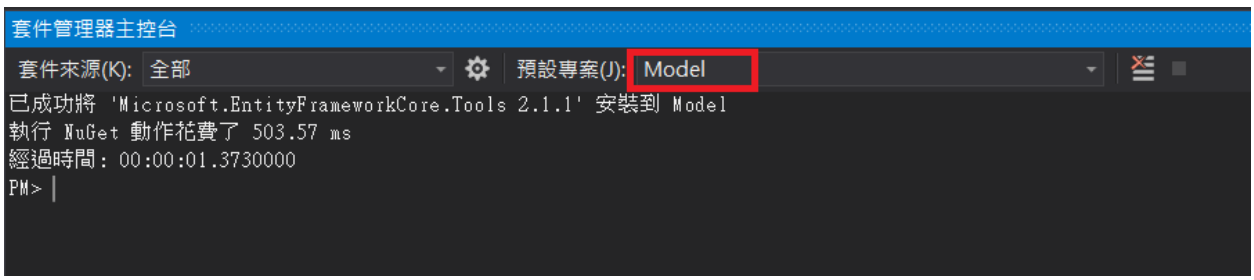
   透過套件管理主控台：

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer -Version 2.1.1 -Project Model
Install-Package Microsoft.EntityFrameworkCore.Tools -Version 2.1.1 -Project Model
```

透過nuget 管理工具（記得安裝在 Model 專案之下）：




4. 透過 EF Core Tools 產生 Model 前需要先成功建置方案，先將剛剛未完成的測試方法註解
5. 使用 EF Core Tools 來產生 Model，開啟套件管理主控台，修改預設專案為 Model




6. 在套件管理主控台輸入，記得更改連線字串為本地資料庫，資料庫為剛剛建立的 DEMO：

```
Scaffold-DbContext "Server=.\sqlexpress;Database=DEMO;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer -Force
```

7. 建立成功後即可在 Model 專案看到對應的 DBContext 及 Model
8. 在DEMOContext中，刪除 OnConfiguring 方法，讓連線字串改以建構式時注入的方式產生

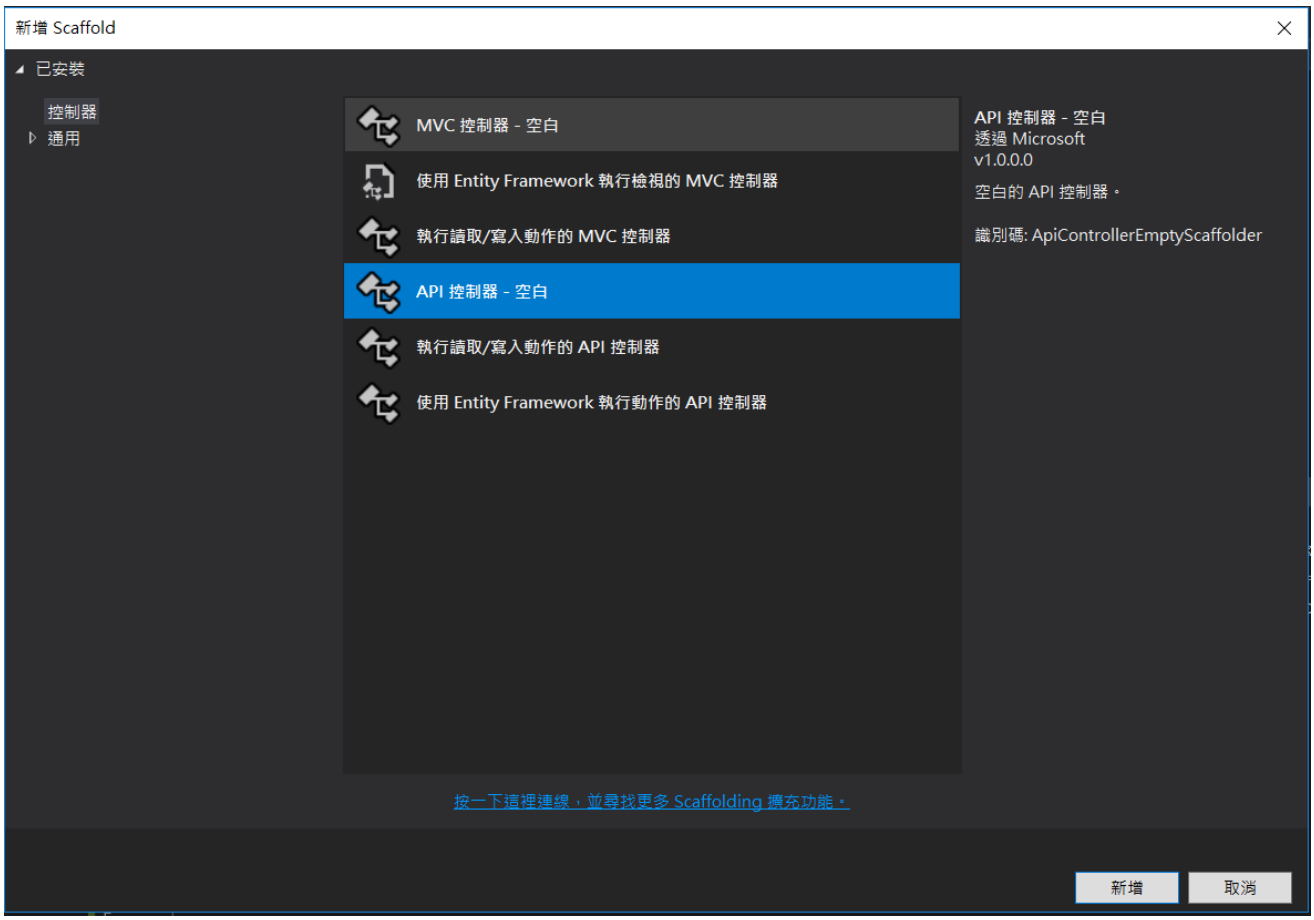
```
public DEMOContext(DbContextOptions<DEMOContext> options)
    : base(options)
{ }
```

9. 回到測試方法，取消註解並引用剛剛產生的 Product，就可以解決部分的編譯錯誤

接著建立測試對象 - controller

## 建立 Controller

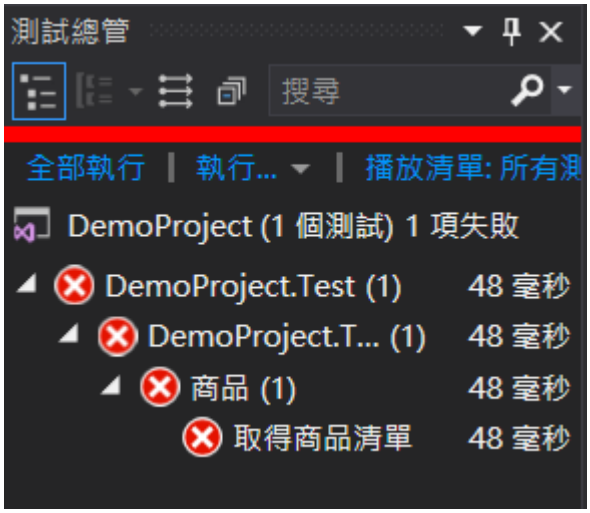
接著透過 .NET Core 建立 Controller，與 MVC 相似，我們在 API 專案下的 Controller 新增 API 控制器，控制器名稱為 ProductsController




接著回到測試加入 ProductsController 的參考。最後，剩下 controller.GetAll() 的方法，直接透過 `ctrl+.` 產生，產生的方法會自動加到 ProductsController 下，我們先不急著實作

## 第一個紅燈

回到測試方法，所有的編譯錯誤都已經解決，我們試著跑一次測試（ `ctrl + R`, `A` ），並在測試總管看到這個結果




紅燈是測試的第一個環節，原因則是因為尚未實作 ProductsController GetAll 方法，至此，我們才要開始開發 production code

*注意流程是先產生 test code，再一一產生對象，與原先先開發再產生 test code 的流程不同*

## 實作功能

現在將焦點放在 GetAll 方法上，在 GetAll 中，僅有一件事：取得並返回商品資料

在 controller 中，我們僅關注返回商品資料這件事情，至於資料怎麼來則交由其他對象處理，這裡我們利用 Repository Pattern 來設計，因此在 GetAll 中實作如下，並修改回傳類型由 object 改為 IEnumerable<Product>：

```
public IEnumerable<Product> GetAll()
{
    var repository = new IRepository<Product>();
    return repository.ReadAll();
}
```

一樣，修改完成後，出現了多處的編譯錯誤，原因是還沒產生對應的對象，接著我們來定義 IRepository 這個介面，另外，我們新增一個 DataSource 專案，將資料處理由 API 層抽離出來，此專案為 .NET Core 類別庫，再在 DataSource 專案下新增一個介面，介面方法如下：

```
public interface IRepository<TEntity>
{
    /// <summary>
    /// 查詢所有資料
    /// </summary>
    /// <returns></returns>
    IEnumerable<TEntity> ReadAll();
}
```

此次專案我們僅需定義 ReadAll 方法

接著實作這個介面，一樣在 DataSource 專案底下，新增一個類別 ProductRepository，並實作 IRepository

```
public class ProductRepository : IRepository<Product>
{
    /// <summary>
    /// 查詢所有資料
    /// </summary>
    /// <returns></returns>
    public IEnumerable<Product> ReadAll()
    {
        private readonly DEMOContext _context;

        public ProductRepository(DEMOContext context)
        {
            _context = context;
        }

        /// <summary>
        /// 查詢所有資料
        /// </summary>
        /// <returns></returns>
        public IEnumerable<Product> ReadAll()
        {
            return _context.Product;
        }
    }
}
```

注意，為了減少相依性，我們用到 .NET Core 提供的相依性注入（DI, Dependency Injection）將 DBContext 注入

既然我們已經用到 DI，那麼回到 ProductsController，我們一樣也用同樣的方式將 Repository 注入，修改 ProductsController 如下，別忘了加入 IRepository 的參考

```
[Produces("application/json")]
[Route("api/Product")]
public class ProductsController : Controller
{
    private readonly IRepository<Product> _repository;

    public ProductsController(IRepository<Product> repository)
    {
        _repository = repository;
    }

    /// <summary>
    /// 取得商品清單
    /// </summary>
    /// <returns></returns>
    public IEnumerable<Product> GetAll()
    {
        return _repository.ReadAll();
    }
}
```

至於該如何在 .NET Core 框架下註冊相關服務，接下來我們將焦點放在 .NET Core 的相依性注入上

### .NET Core 相依性注入

.NET Core 的相依性注入流程，參考這篇文章有詳細的介紹：https://ithelp.ithome.com.tw/articles/10193172

首先，在 Startup.cs 的 ConfigureServices 註冊 DBContext

```
services.AddDbContext<DEMOContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
```

注意到我們的連線字串是透過 config 設置來取得，與 .NET Framework 的 web.config / app.config 類似，我們可以在專案中的 appsettings.json 加入連線字串，記得修改連線字串

```
"ConnectionStrings": {
    "DefaultConnection":
"Server=.\sqlexpress;Database=DEMO;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
```

> 設定檔可以使用 .config 或 .json

如此便完成 DBContext 的註冊，接著在需要使用的時候，透過建構式注入即可

接著將註冊 Repository

```
services.AddScoped<IRepository<Product>, ProductRepository>();
```

我們使用 Scoped 方式，讓該實體存活在一次完整的 request 中

## NSubstitute

我們已經將所需資料服務改以 DI 的方式注入到 controller 中，回到測試方法修改相應的做法，需要將 IRepository 也注入到 ProductsController 中，但若是直接使用 IRepository 來取得資料，則勢必與資料庫連線，不符合「單元測試」的原則，因此我們必須模擬一個 IRepository 的實作來獲取資料，這樣一個模擬在測試中我們稱為 Stub，我們需要再產生一個 Stub 類別，該類別實作介面（也就是 IRepository），並讓此類別的方法專注於直接回傳我們需要得到的內容，如下：

```
public class StubProductRepository : IRepository<Product>
{
    public IEnumerable<Product> ReadAll()
    {
        return new List<Product> { ... };
    }
}
```

但此類別僅供測試用，測試時又需要去產生每一個介面的模擬類別，因此我們改透過 NSubstitute，使用下列方法產生一個介面實作物件，並在之後指定該 Stub 回傳的結果：

```
var stubRepository = Substitute.For<IRepository<Product>>();
stubRepository.ReadAll().Returns(預計回傳的結果);
```

修改後的測試方法如下：

```
[TestMethod]
public void 取得商品清單()
{
    // Arrange
    var expected = new List<Product>
    {
        new Product { Id = 1, Name = "科幻四小俠", Price = 299, InStock = 6, Brief = "垃圾書刊"
},
        new Product { Id = 2, Name = "宇宙科幻", Price = 560, InStock = 2, Brief = "暢銷書籍" },
        new Product { Id = 3, Name = "名偵探-ㄎㄅ", Price = 250, InStock = 5, Brief = "都在唬爛"
},
        new Product { Id = 4, Name = "這不是我認識的偵探", Price = 520, InStock = 0, Brief =
"._." }
    };

    // 這裡設定 Stub 及回傳內容
    var stubRepository = Substitute.For<IRepository<Product>>();
    stubRepository.ReadAll().Returns(expected);

    var controller = new ProductsController(stubRepository);

    // Act
    var actual = controller.GetAll();

    // Assert
    Assert.AreEqual(expected, actual);
}
```

## 第一個綠燈

到目前為止，已經完成 test code 的撰寫，且依照單元測試的原則實作 production code，按下 `ctrl + R, A` 來執行所有測試（或是 `ctrl + R, T` 執行單一測試），可以看到得到綠燈的結果，即完成這次的 TDD 開發流程，往後不管進行重構或是邏輯修改，別忘了再開始前先執行所有測試，並養成每一次更改都按下 `ctrl + R, A` 的習慣

可以在完成一個功能後馬上在 git 上 commit 一個版本，好處是我們可以確保這次的 commit 是沒有問題的，若是後續改壞，我們還可以回到這次的 commit

## 下一個功能

繼續完成剩下的兩個測試，先專注在「取得所選商品總計金額」，簡單的抽象步驟如下：

```
[TestMethod]
public void 取得所選商品總計金額()
{
    // Arrange
    var input = new List<Product>
    {
        new Product { Id = 1 },
        new Product { Id = 2 }
    };

    var expected = 859m;
    var stubRepository = Substitute.For<IRepository<Product>>();
    stubRepository.ReadRowsByIds(Arg.Any<List<Product>>()).Returns(input);
    var controller = new ProductsController(stubRepository);

    // Act
    var actual = controller.GetSum(input);

    // Assert
    Assert.AreEqual(expected, actual);
}
```

注意中間的 stubRepository.ReadRows(Arg.Any<List<Product>>())...，因為我們僅須關注回傳的結果，所以利用 Arg.Any<T> 針對傳入的任何參數皆回傳同樣的值

完成抽象的測試方法後，一樣會看到編譯錯誤的方法，我們將該方法補上：

```
/// <summary>
/// 取得總計金額
/// </summary>
/// <param name="products">選擇的商品</param>
/// <returns></returns>
public decimal GetSum(List<Product> products)
{
    throw new System.NotImplementedException();
}
```

由於我們新增了一個抽象方法 ReadRows，因此需要在 IRepository 介面以及他的實作類別 ProductRepository 下再新增此方法：

IRepository.cs

```
/// <summary>
/// 查詢特定資料 (By ids)
/// </summary>
/// <returns></returns>
IEnumerable<TEntity> ReadRowsByIds(List<TEntity> condition);
```

ProductRepository.cs

```
/// <summary>
/// 查詢特定資料 (By ids)
/// </summary>
/// <param name="condition">查詢條件</param>
/// <returns></returns>
public IEnumerable<Product> ReadRowsByIds(List<Product> condition)
{
    throw new System.NotImplementedException();
}
```

因為我們已經修改了相關的程式碼，為了避免先前成功的測試搞壞，執行全部測試，得到現在這個測試方法紅燈的結果，接著我們進行實作，如下：

ProductsController.cs

```
/// <summary>
/// 取得總計金額
/// </summary>
/// <param name="products">選擇的商品</param>
/// <returns></returns>
public decimal GetSum(List<Product> products)
{
    var selected = _repository.ReadRowsByIds(products);
    return selected.Sum(r => r.Price);
}
```

> 之後若要實際進行整合測試，別忘記加上 [HttpPost]、[Route("sum")]，確保 POST product/sum 能對應到正確的方法，以及在參數中加上 [FromBody] 或是 [FromForm] 指定資料來源

ProductRepository.cs

```
/// <summary>
/// 查詢特定資料 (By ids)
/// </summary>
/// <param name="condition">查詢條件</param>
/// <returns></returns>
public IEnumerable<Product> ReadRowsByIds(List<Product> condition)
{
    return _context.Product.Where(r => condition.Select(c => c.Id).Contains(r.Id));
}
```

執行全部測試，得到綠燈，結束，commit，接著最後一個功能

## 最後一個功能

*當然，需求永遠沒有最後*

最後一個功能**取得所選商品總計金額_未選擇商品**，我們假設使用者未選擇任何商品即呼叫 GetSum API（此處為了測試，通常前端也會做防呆避免使用者呼叫），應該在 API 就判斷使用者未選擇資料而不執行後續的邏輯，因此測試方法如下：

```
[TestMethod]
public void 取得所選商品總計金額_未選擇商品()
{
    // Arrange
    var input = new List<Product>();

    var expected = 0;
    var stubRepository = Substitute.For<IRepository<Product>>();
    var controller = new ProductsController(stubRepository);

    // Act
    var actual = controller.GetSum(input);

    // Assert
    stubRepository.DidNotReceive().ReadRowsByIds(Arg.Any<List<Product>>());
    Assert.Equals(expected, actual);
}
```

注意，這裡除了驗證回傳的結果之外，依據我們設定，也驗證了 ReadRowsByIds 是否被執行

接著執行測試，再次得到紅燈，接著就要修改 production code

*修改功能別忘記隨時執行測試*

我們將 GetSum 方法修改如下：

```
/// <summary>
/// 取得總計金額
/// </summary>
/// <param name="products">選定的商品</param>
/// <returns></returns>
public decimal GetSum(List<Product> products)
{
    // 未選擇商品
    if (products.Count() == 0)
        return 0;
    else
    {
        var selected = _repository.ReadRowsByIds(products);
        return selected.Sum(r => r.Price);
    }
}
```

再次執行測試，得到所有測試方法皆為綠燈的結果，到這裡，我們已經利用 TDD 的開發流程完成了所有功能