RESTful Day #7: Unit Testing and Integration Testing in WebAPI using NUnit and Moq framework (Part1)-Akhil Mittal.

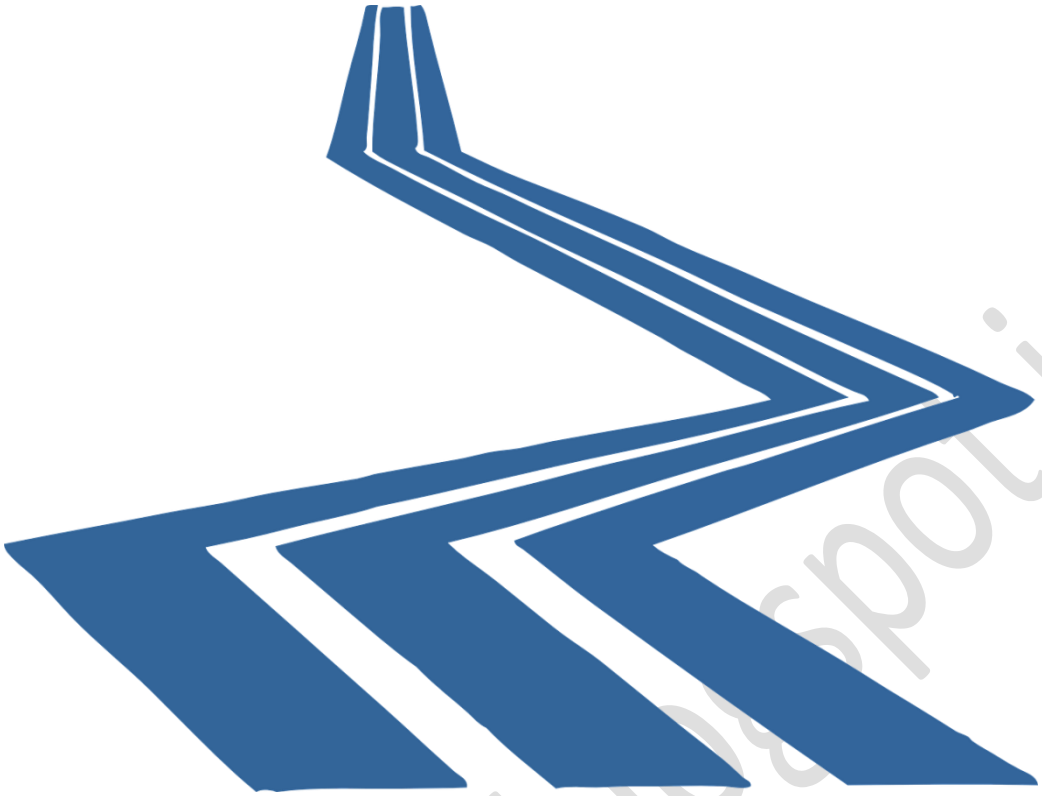
Table of Contents

Introduction

We have been learning a lot in WebAPI. We covered almost all the techniques required to build a robust and a full stack REST service using asp.net WebAPI, right from creating a service to making it a secure and ready to use boilerplate with enterprise level applications. In this article we'll learn on how to focus on test driven development and write unit tests for our service endpoints and business logic. I'll use NUnit and Moq framework to write test cases for business logic layer and controller methods. I'll cover less theory and focus more on practical implementations on how to use these frameworks to write unit tests. I have segregated the article into two parts. First part focusses on testing business logic and class libraries created as BusinessServices in our code base. Second part will focus on testing a Web API. The purpose of segregation is simple; the scope of this article is very large and may turn up into a very large post which would be not easy to read in a go.

Roadmap

Following is the roadmap I have setup to learn WebAPI step by step,

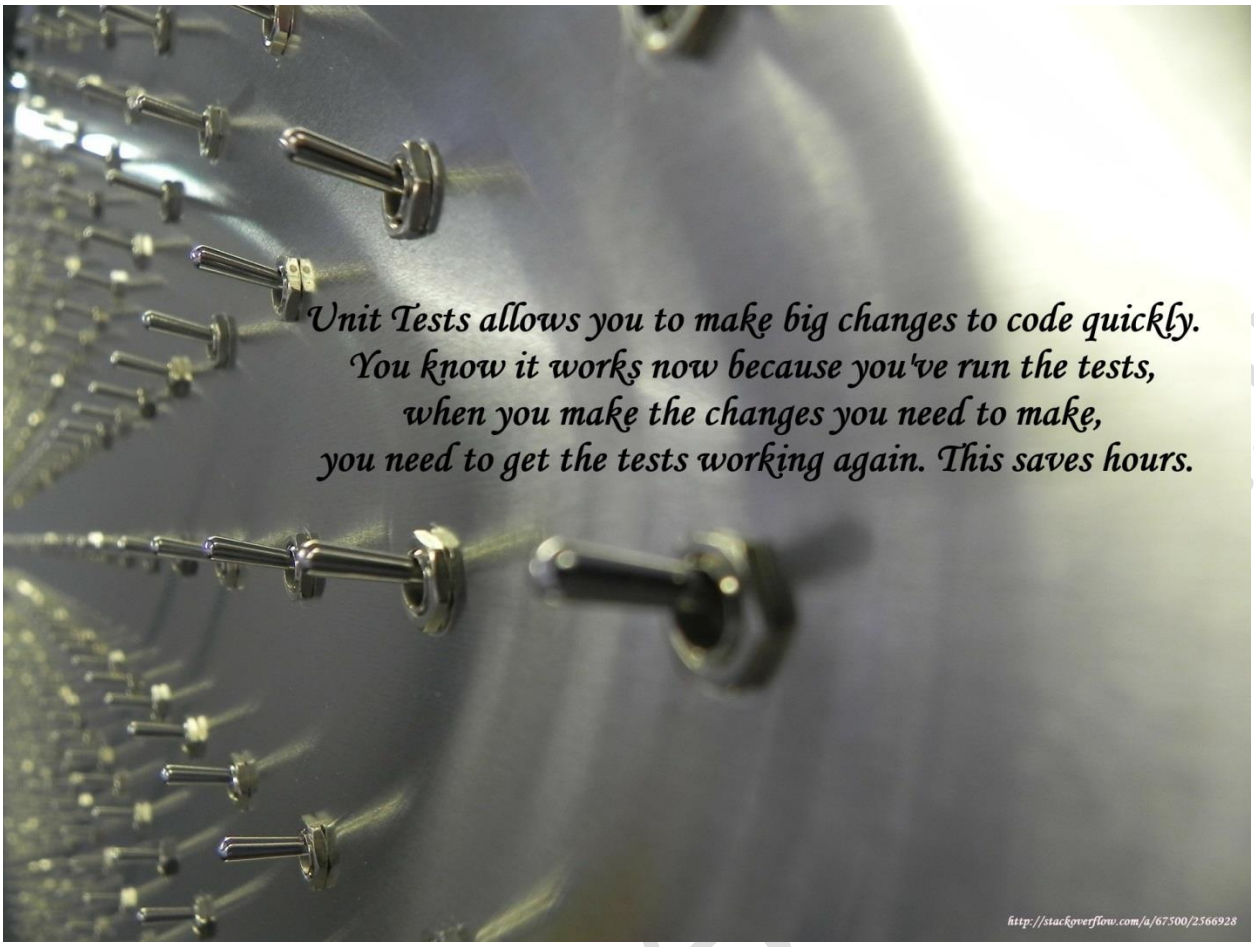


- RESTful Day #1: Enterprise level application architecture with Web APIs using Entity Framework, Generic Repository pattern and Unit of Work.
- RESTful Day #2: Inversion of control using dependency injection in Web APIs using Unity Container and Bootstrapper.
- RESTful Day #3: Resolve dependency of dependencies using Inversion of Control and dependency injection in Asp.net Web APIs with Unity Container and Managed Extensibility Framework (MEF).
- RESTful Day #4: Custom URL Re-Writing/Routing using Attribute Routes in MVC 4 Web APIs.
- RESTful Day #5: Basic Authentication and Token based custom Authorization in Web APIs using Action Filters.
- RESTful Day #6: Request logging and Exception handing/logging in Web APIs using Action Filters, Exception Filters and NLog.
- RESTful Day #7: Unit Testing and Integration Testing in WebAPI using NUnit and Moq framework (Part1).
- RESTful Day #8: Unit Testing and Integration Testing in WebAPI using NUnit and Moq framework (Part 2).
- RESTful Day #9: Extending OData support in ASP.NET Web APIs.

I'll purposely use Visual Studio 2010 and .net Framework 4.0 because there are few implementations that are very hard to find in .Net Framework 4.0, but I'll make it easy by showing how we can do it.

Unit Tests

"Unit tests allow you to make big changes to code quickly. You know it works now because you've run the tests, when you make the changes you need to make, you need to get the tests working again. This saves hours." I got this from a post at stack overflow, and I completely agree to this statement.



A good unit test helps a developer to understand his code and most importantly the business logic. Unit tests help to understand all the aspects of business logic, right from the desired input and output to the conditions where the code can fail. A code having a well written unit tests have very less chances to fail provided the unit tests cover all the test cases required to execute.

NUnit

There are various frameworks available for Unit tests. NUnit is the one that I prefer. NUnit gels well with .Net and provide flexibility to write unit tests without hassle. It has meaningful and self-explanatory properties and class names that help developer to write the tests in an easy way. NUnit provides an easy to use interactive GUI where you can run the tests and get the details .It shows the number of tests passed or fail in a beautiful fashion and also gives the stack trace in case any test fails, thereby enabling you to perform the first level of debugging at the GUI itself. I suggest downloading and installing NUnit on your machine for running the tests. We'll use NUnit GUI after we write all the tests. I normally use inbuilt GUI of NUnit provided by Re-sharper integrated in my Visual Studio. Since Re-sharper is a paid library only few developers may have it integrated, so I suggest you to use NUnit GUI to run the tests. Since we are using Visual Studio 2010, we need to use the older version of NUnit i.e. 2.6.4. You can download and run the .msi and install on your machine following this URL.

Downloads

Current Release: NUnit 3.0.1

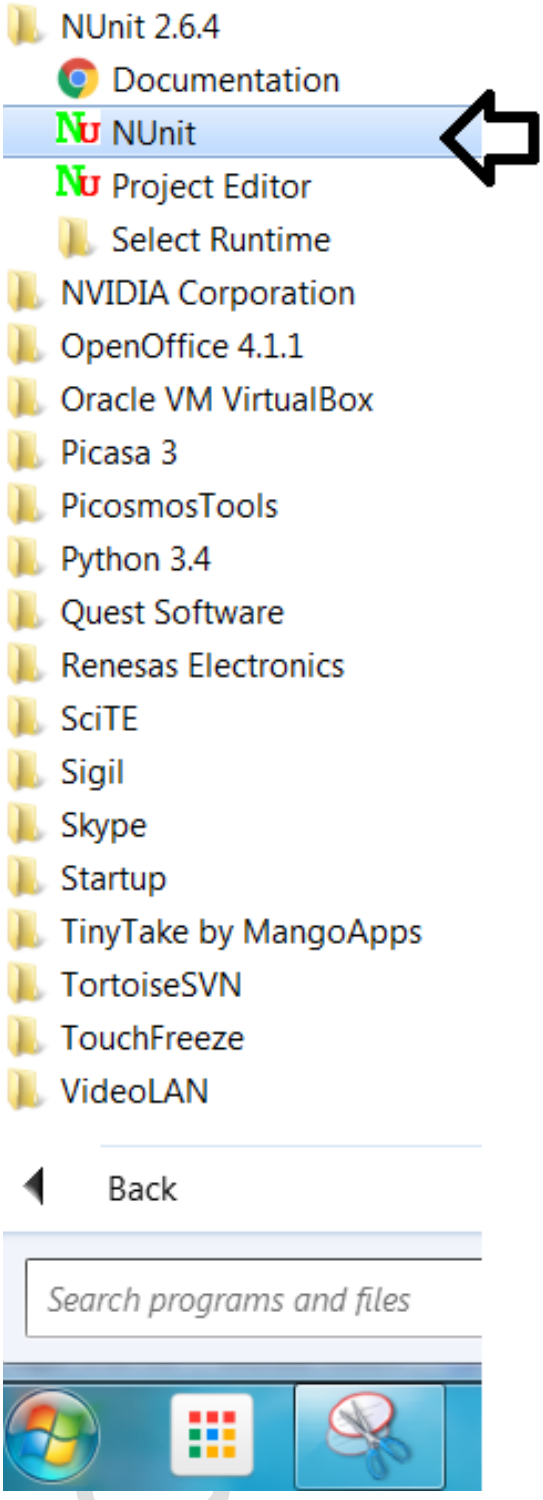
NUnit 3.0.1 - December 1, 2015	
win	NUnit.3.0.1.msi
bin	NUnit-3.0.1.zip
src	NUnit-3.0.1-src.zip
sl	NUnitSL-3.0.1.zip
cf	NUnitCF-3.0.1.zip

Additional downloads for NUnit 3.0 can be found on the GitHub releases pages.

Previous Release: NUnit 2.6.4

NUnit 2.6.4 - December 16, 2014	
win	NUnit-2.6.4.msi
bin	NUnit-2.6.4.zip
win .net 1.1	NUnit-2.6.4-net-1.1.msi
bin .net 1.1	NUnit-2.6.4-net-1.1.zip
src	2.6.4.zip
doc	NUnit-2.6.4-docs.zip

Once you finish installation, you'll see NUnit installed in your installed items on your machine as shown in below image,



Moq Framework

Moq is a simple and straight forward library to mock the objects in C#. We can mock data, repositories classes and instances with the help of mock library. So when we write unit tests, we do not execute them on the actual class instances, instead perform in-memory unit testing by making proxy of class objects. Like NUnit,

Moq library classes are also easy to use and understand. Almost all of its methods, classes and interfaces names are self-explanatory.

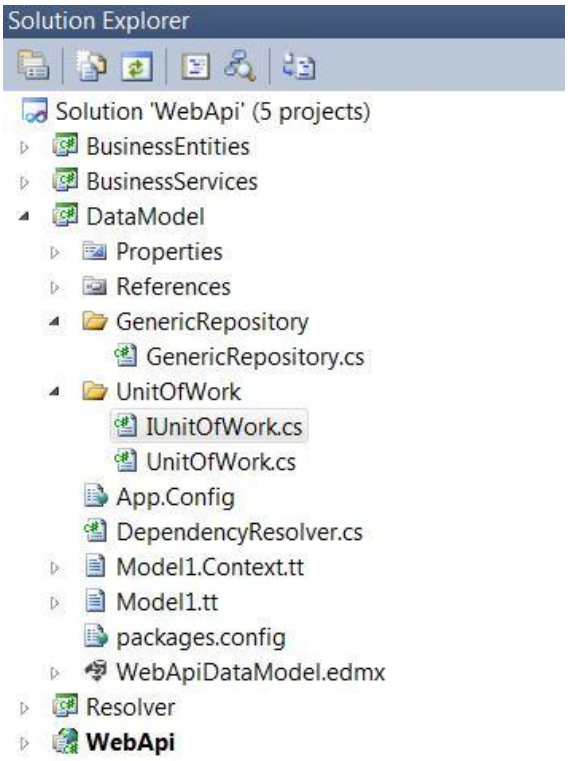
Following is the list taken from Wikipedia on why to use mock objects,

- The object supplies non-deterministic results (e.g., the current time or the current temperature);
- Has states that are not easy to create or reproduce (e.g., a network error);
- Is slow (e.g., a complete database, which would have to be initialized before the test);
- Does not yet exist or may change behavior;
- Would have to include information and methods exclusively for testing purposes (and not for its actual task).

So whatever test we write, we actually execute that on test data and proxy objects i.e. not the instances of real classes. We'll use Moq to mock data and repositories so that we do not hit database again and again for executing unit tests. You can read more about Moq in this article.

Setup Solution

I'll use this article to explain how to write unit tests for business logic i.e. covering our business logic layer and for WebAPI controllers. The scope of Unit tests should not be only limited to business logic or endpoints but should spread over all publically exposed logics like filters and handlers as well. Well written unit tests should cover almost all the code. One can track the code coverage through some of the tools available online. We'll not test filters and common classes but will focus on controllers and business logic layer and get an idea of how to proceed with unit tests. I'll use the same source code that we used till Day# 6 of the series and will proceed with the latest code base that we got out of last article of the series. Code base is available for download with this post. When you take the code base from my last article and open it in visual studio, you'll see the project structure something like as shown in below image,



IUnitOfWork is the new interface that I have added just to facilitate interface driven development. It helps in mocking objects and improved structure and readability. Just open the visual studio and add a new interface named IUnitOfWork under UnitOfWork folder in DataModel project and define the properties used in UnitOfWork class as shown below,

```
public interface IUnitOfWork
{
    #region Properties
    GenericRepository<Product> ProductRepository { get; }
    GenericRepository<User> UserRepository { get; }
    GenericRepository<Token> TokenRepository { get; }
    #endregion

    #region Public methods
    /// <summary>
    /// Save method.
    /// </summary>
    void Save();
    #endregion
}
```

Now, go to UnitOfWork class and inherit that class using this interface, so UnitOfWork class becomes something like this,

```
#region Using Namespaces...

using System;
using System.Collections.Generic;
using System.Data.Entity;
```

```
using System.Diagnostics;
using System.Data.Entity.Validation;
using DataModel.GenericRepository;

#endregion

namespace DataModel.UnitOfWork
{
    /// <summary>
    /// Unit of Work class responsible for DB transactions
    /// </summary>
    public class UnitOfWork : IDisposable, IUnitOfWork
    {
        #region Private member variables...

        private readonly WebApiDbEntities _context = null;
        private GenericRepository<User> _userRepository;
        private GenericRepository<Product> _productRepository;
        private GenericRepository<Token> _tokenRepository;
        #endregion

        public UnitOfWork()
        {
            _context = new WebApiDbEntities();
        }

        #region Public Repository Creation properties...

        /// <summary>
        /// Get/Set Property for product repository.
        /// </summary>
        public GenericRepository<Product> ProductRepository
        {
            get
            {
                if (this._productRepository == null)
                    this._productRepository = new GenericRepository<Product>(_context);
                return _productRepository;
            }
        }

        /// <summary>
        /// Get/Set Property for user repository.
        /// </summary>
        public GenericRepository<User> UserRepository
        {
            get
            {
                if (this._userRepository == null)
                    this._userRepository = new GenericRepository<User>(_context);
                return _userRepository;
            }
        }

        /// <summary>
        /// Get/Set Property for token repository.
```

```
        /// </summary>
        public GenericRepository<Token> TokenRepository
        {
            get
            {
                if (this._tokenRepository == null)
                    this._tokenRepository = new GenericRepository<Token>(_context);
                return _tokenRepository;
            }
        }
        #endregion

        #region Public member methods...
        /// <summary>
        /// Save method.
        /// </summary>
        public void Save()
        {
            try
            {
                _context.SaveChanges();
            }
            catch (DbEntityValidationException e)
            {

                var outputLines = new List<string>();
                foreach (var eve in e.EntityValidationErrors)
                {
                    outputLines.Add(string.Format("{0}: Entity of type \"{1}\" in state
\"{2}\" has the following validation errors:", DateTime.Now, eve.Entry.Entity.GetType().Name,
eve.Entry.State));
                    foreach (var ve in eve.ValidationErrors)
                    {
                        outputLines.Add(string.Format("- Property: \"{0}\", Error: \"{1}\"",
ve.PropertyName, ve.ErrorMessage));
                    }
                }
                System.IO.File.AppendAllLines(@"C:\errors.txt", outputLines);

                throw e;
            }

        }

        #endregion

        #region Implementing IDiosposable...

        #region private dispose variable declaration...
        private bool disposed = false;
        #endregion

        /// <summary>
        /// Protected Virtual Dispose method
        /// </summary>
        /// <param name="disposing"></param>
```

```
        protected virtual void Dispose(bool disposing)
        {
            if (!this.disposed)
            {
                if (disposing)
                {
                    Debug.WriteLine("UnitOfWork is being disposed");
                    _context.Dispose();
                }
            }
            this.disposed = true;
        }

        /// <summary>
        /// Dispose method
        /// </summary>
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }
        #endregion
    }
}
```

So, now all the interface members defined in IUnitOfWork are implemented in UnitOfWork class,

```
public interface IUnitOfWork
    {
        #region Properties
        GenericRepository<Product> ProductRepository { get; }
        GenericRepository<User> UserRepository { get; }
        GenericRepository<Token> TokenRepository { get; }
        #endregion

        #region Public methods
        /// <summary>
        /// Save method.
        /// </summary>
        void Save();
        #endregion
    }
```

Doing this will not change the functionality of our existing code, but we also need to update the business services with this Interface. We'll pass this IUnitOfWork interface instance inside services constructors instead of directly using UnitOfWork class.

```
        private readonly IUnitOfWork _unitOfWork;

        public ProductServices(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }
```

So our User service, Token service and product service constructors becomes as shown below,

Product Service:

```
private readonly IUnitOfWork _unitOfWork;

/// <summary>
/// Public constructor.
/// </summary>
public ProductServices(IUnitOfWork unitOfWork)
{
    _unitOfWork = unitOfWork;
}
```

User Service:

```
private readonly IUnitOfWork _unitOfWork;

/// <summary>
/// Public constructor.
/// </summary>
public UserServices(IUnitOfWork unitOfWork)
{
    _unitOfWork = unitOfWork;
}
```

Token Service:

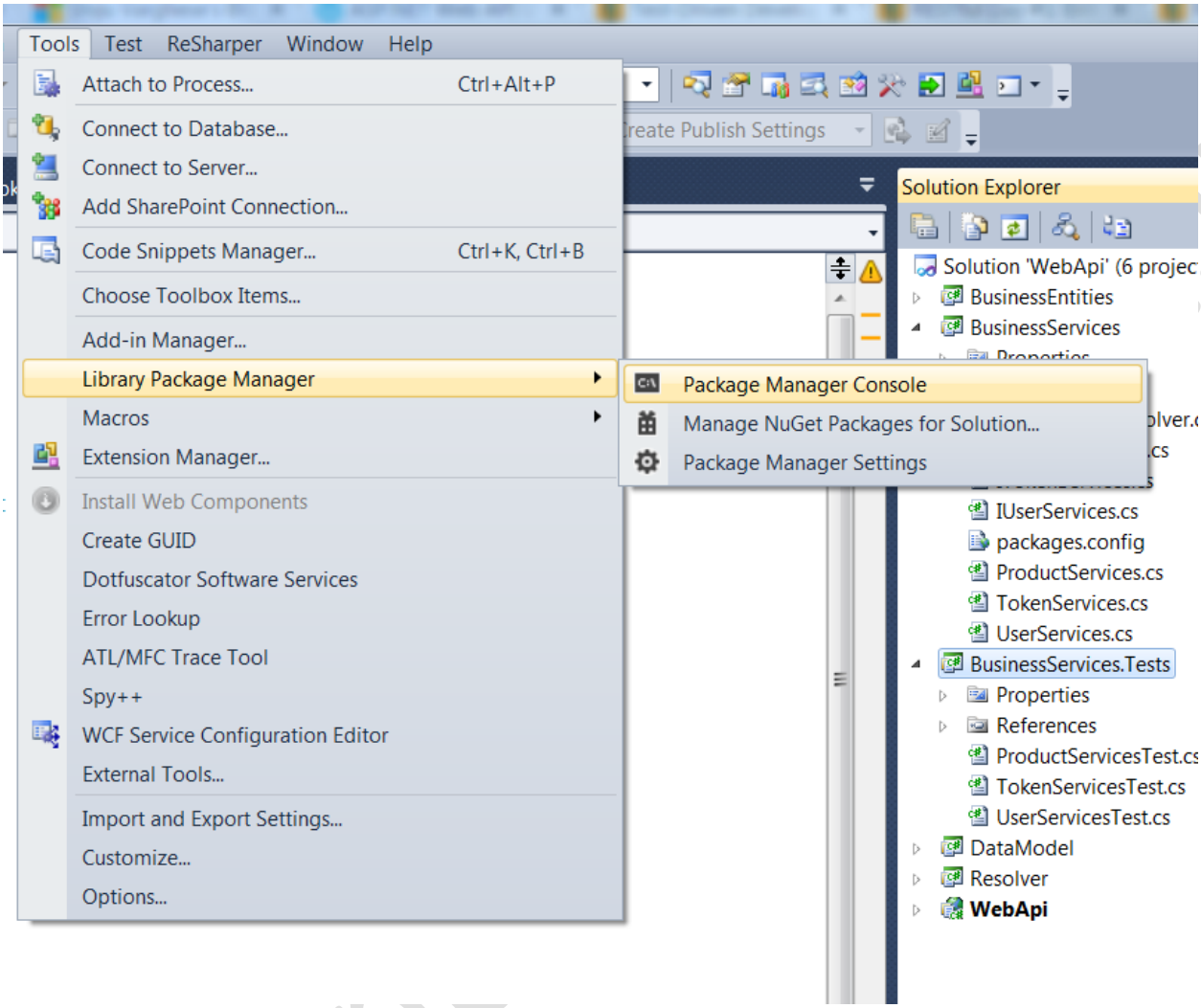
```
#region Private member variables.
 private readonly IUnitOfWork _unitOfWork;
#endregion

#region Public constructor.
/// <summary>
/// Public constructor.
/// </summary>
public TokenServices(IUnitOfWork unitOfWork)
{
    _unitOfWork = unitOfWork;
}
#endregion
```

Testing Business Services

We'll start writing unit tests for BusinessServices project.

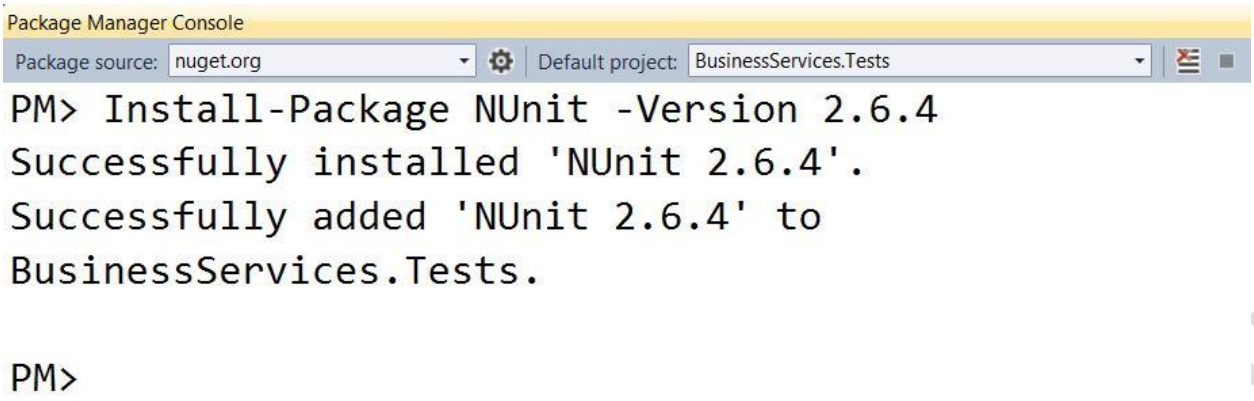
Step 1: Test Project

Add a simple class library in the existing visual studio and name it BusinessServices.Tests. Open Tools->Library Packet Manager->Packet manager Console to open the package manager console window. We need to install come packages before we proceed.

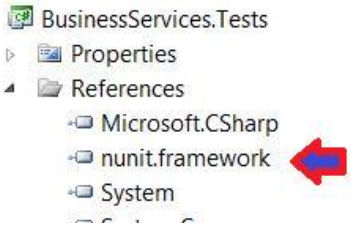


Step 2: Install NUnit package

In package manager console, select BusinessServices.Tests as default project and write command "**Install-Package NUnit –Version 2.6.4**". If you do not mention the version, the PMC (Package manage Console) will try to download the latest version of NUnit nugget package but we specifically need 2.6.4, so we need to mention the version. Same applies to when you try to install any such package from PMC

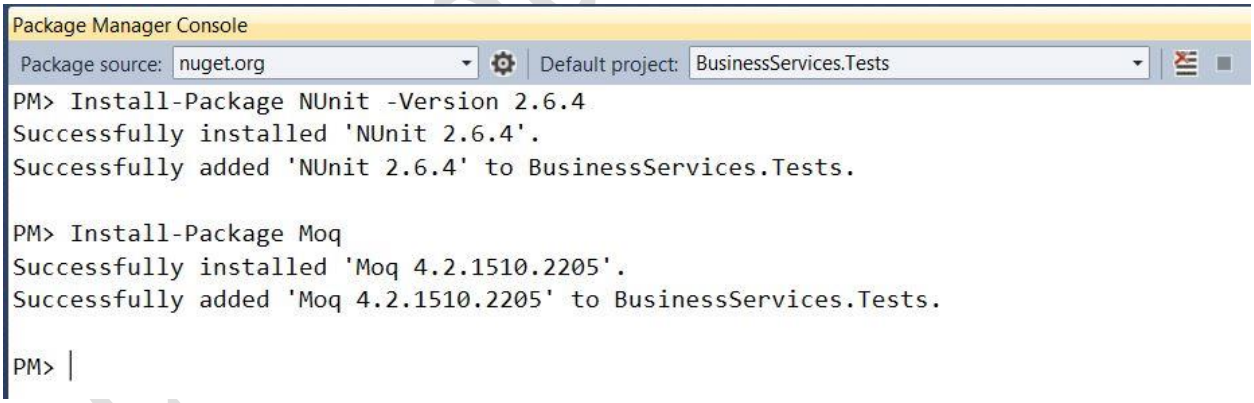


After successfully installed, you can see the dll reference in project references i.e. nunit.framework,

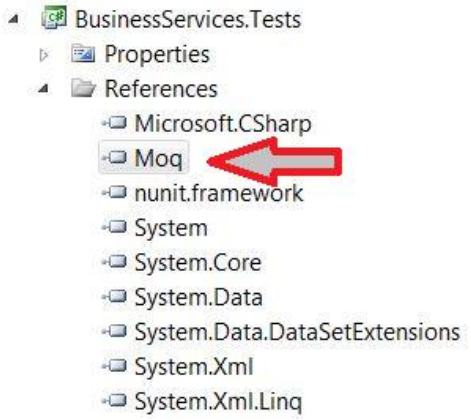


Step 3: Install Moq framework

Install the framework on the same project in the similar way as explained in Step 2. Write command "**Install-Package Moq**" .Here we use latest version of Moq



Therefore added dll,



Step 4: Install Entity Framework

Install-Package EntityFramework –Version 5.0.0

Step 5: Install AutoMapper

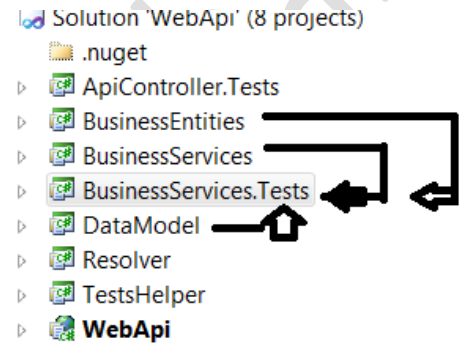
Install-Package AutoMapper –Version 3.3.1

Our package.config i.e. automatically added in the project looks like,

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="AutoMapper" version="3.3.1" targetFramework="net40" />
  <package id="EntityFramework" version="5.0.0" targetFramework="net40" />
  <package id="Moq" version="4.2.1510.2205" targetFramework="net40" />
  <package id="NUnit" version="2.6.4" targetFramework="net40" />
</packages>
```

Step 6: References

Add references of DataModel, BusinessServices , BusinessEntities project to this project.



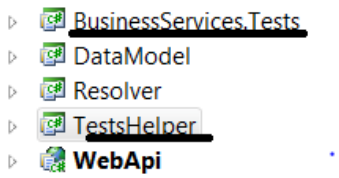
TestHelper

We will require few helper files that would be needed in BusinessServices.Tests project and in our WebAPI.Tests project that we'll create later. To place all the helper files, I have created one more class library project named TestHelper. Just right click the solution and add new project named TestHelper and add a class named DataInitializer.cs into it. This class contains three simple methods to fetch i.e. User's, Product's and Token's dummy data. You can use following code as the class implementation,

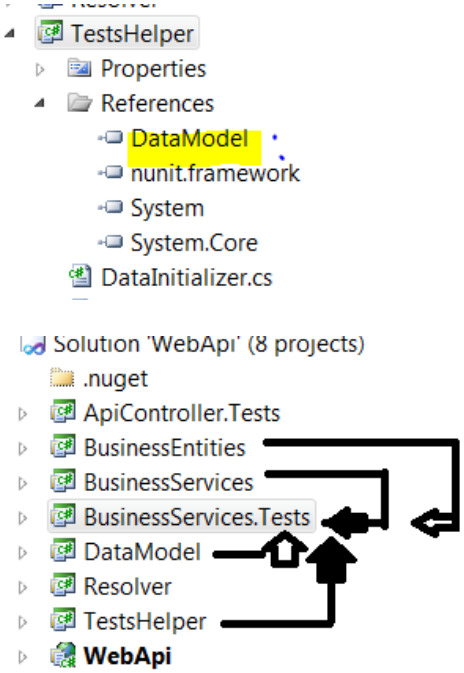
```
using System;
using System.Collections.Generic;
using DataModel;

namespace TestsHelper
{
/// <summary>
/// Data initializer for unit tests
/// </summary>
public class DataInitializer
{
/// <summary>
/// Dummy products
/// </summary>
/// <returns></returns>
public static List<Product> GetAllProducts()
{
var products = new List<Product>
{
new Product() {ProductName = "Laptop"},
new Product() {ProductName = "Mobile"},
new Product() {ProductName = "HardDrive"},
new Product() {ProductName = "IPhone"},
new Product() {ProductName = "IPad"}
};
return products;
}

/// <summary>
/// Dummy tokens
/// </summary>
/// <returns></returns>
public static List<Token> GetAllTokens()
{
var tokens = new List<Token>
{
new Token()
{
AuthToken = "9f907bdf-f6de-425d-be5b-b4852eb77761",
ExpiresOn = DateTime.Now.AddHours(2),
IssuedOn = DateTime.Now,
UserId = 1
},
new Token()
{
AuthToken = "9f907bdf-f6de-425d-be5b-b4852eb77762",
ExpiresOn = DateTime.Now.AddHours(1),
IssuedOn = DateTime.Now,
```

```
UserId = 2
}
};

return tokens;
}

/// <summary>
/// Dummy users
/// </summary>
/// <returns></returns>
public static List<User> GetAllUsers()
{
var users = new List<User>
{
new User()
{
UserName = "akhil",
Password = "akhil",
Name = "Akhil Mittal",
},
new User()
{
UserName = "arsh",
Password = "arsh",
Name = "Arsh Mittal",
},
new User()
{
UserName = "divit",
Password = "divit",
Name = "Divit Agarwal",
}
};

return users;
}

}
}
```

In the above class GetAllUsers() fetches dummy data for users, GetAllProducts() fetches dummy data for Products and GetAllTokens() method fetches dummy data for Tokens. So now, our solution has two new projects as shown below,



Add DataModel project reference to TestHelper project and TestHelper project reference to BusinessServices.Tests project.



ProductService Tests

We'll start with setting up the project and setting up the pre-requisites for tests and gradually move on to actual tests.

Tests Setup

We'll proceed with creating ProductServices tests. Add a new class named ProductServicesTests.cs in BusinessServices.Tests project.

Declare variables

Define the private variables that we'll use in the class to write tests,

```
#region Variables
private IProductServices _productService;
private IUnitOfWork _unitOfWork;
private List<Product> _products;
private GenericRepository<Product> _productRepository;
private WebApiDbEntities _dbEntities;
#endregion
```

Variable declarations are self-explanatory where _productService will hold mock for ProductServices, _unitOfWork for UnitOfWork class, _products will hold dummy products from DataInitializer class of TestHelper project, _productRepository and _dbEntities holds mock for Product Repository and WebAPIDbEntities from DataModel project respectively

Write Test Fixture Setup

Test fixture setup is written as a onetime setup for all the tests. It is like a constructor in terms of classes. When we start executing setup, this is the first method to be executed. In this method we'll populate the dummy products data and decorate this method with the [TestFixtureSetUp] attribute at the top that tells compiler that the particular method is a TestFixtureSetup. [TestFixtureSetUp] attribute is the part of NUnit framework, so include it in the class as a namespace i.e. using NUnit.Framework;. Following is the code for TestFixtureSetup.

```
#region Test fixture setup

 /// <summary>
 /// Initial setup for tests
 /// </summary>
 [TestFixtureSetUp]
 public void Setup()
 {
     _products = SetUpProducts();
 }

 #endregion

private static List<Product> SetUpProducts()
 {
     var prodId = new int();
     var products = DataInitializer.GetAllProducts();
     foreach (Product prod in products)
         prod.ProductId = ++prodId;
     return products;

 }
```

SetUpproducts() method fetches products from DataInitializer class and not from database.It also and assigns a unique id to each product by iterating them. The result data is assigned to _products list to be used in setting up mock repository and in every individual test for comparison of actual vs resultant output.

Write Test Fixture Tear Down

Unlike TestFixtureSetup, tear down is used to de-allocate or dispose the objects. It also executes only one time when all the tests execution ends. In our case we'll use this method to nullify _products instance. The attribute used for Test fixture tear down is [TestFixtureTearDown].

Following is the code for teardown.

```
#region TestFixture TearDown.

 /// <summary>
 /// TestFixture teardown
 /// </summary>
 [TestFixtureTearDown]
 public void DisposeAllObjects()
```

```
{
    _products = null;
}

#endregion
```

Note that we have till now not written any unit test.

Write Test Setup

TestFixtureSetUp is a onetime run process where as [SetUp] marked method is executed after each test. Each test should be independent and should be tested with a fresh set of input. Setup helps us to re-initialize data for each test. Therefore all the required initialization for tests are written in this particular method marked with [SetUp] attribute. I have written few methods and initialized the private variables in this method. These lines of code execute after each test ends, so that individual tests do not depend on any other written test and do not get hampered with other tests pass or fail status. Code for Setup,

```
#region Setup
/// <summary>
/// Re-initializes test.
/// </summary>
[SetUp]
public void ReInitializeTest()
{
    _dbEntities = new Mock<WebApiDbEntities>().Object;
    _productRepository = SetUpProductRepository();
    var unitOfWork = new Mock<IUnitOfWork>();
    unitOfWork.SetupGet(s => s.ProductRepository).Returns(_productRepository);
    _unitOfWork = unitOfWork.Object;
    _productService = new ProductServices(_unitOfWork);
}

#endregion
```

We make use of Mock framework in this method to mock the private variable instances. Like for _dbEntities we write `_dbEntities = new Mock<WebApiDbEntities>().Object;` . This means that we are mocking WebDbEntities class and getting its proxy object. Mock class is the class from Moq framework, so include the respective namespace `using Moq;` in the class

Write Test Tear down

Like test Setup runs after every test, similarly Test [TearDown] is invoked after every test execution is complete. You can use tear down to dispose and nullify the objects that are initialized while setup. The method for tear down should be decorated with [TearDown] attribute. Following is the test tear down implementation.

```
/// <summary>
/// Tears down each test data
```

```
        /// </summary>
        [TearDown]
        public void DisposeTest()
        {
            _productService = null;
            _unitOfWork = null;
            _productRepository = null;
            if (_dbEntities != null)
                _dbEntities.Dispose();
        }
```

Mocking Repository

I talked about mocking repository for the entities. I have created a method SetUpProductRepository() to mock Product Repository and assign it to _productrepository in ReInitializeTest() method.

```
private GenericRepository<Product> SetUpProductRepository()
{

        // Initialise repository
        var mockRepo = new Mock<GenericRepository<Product>>(MockBehavior.Default, _dbEntities);

        // Setup mocking behavior
        mockRepo.Setup(p => p.GetAll()).Returns(_products);

        mockRepo.Setup(p => p.GetByID(It.IsAny<int>()))
        .Returns(new Func<int, Product>(
        id => _products.Find(p => p.ProductId.Equals(id))));

        mockRepo.Setup(p => p.Insert((It.IsAny<Product>())))
        .Callback(new Action<Product>(newProduct =>
        {
        dynamic maxProductID = _products.Last().ProductId;
        dynamic nextProductID = maxProductID + 1;
        newProduct.ProductId = nextProductID;
        _products.Add(newProduct);
        }));

        mockRepo.Setup(p => p.Update(It.IsAny<Product>()))
        .Callback(new Action<Product>(prod =>
        {
        var oldProduct = _products.Find(a => a.ProductId == prod.ProductId);
        oldProduct = prod;
        }));

        mockRepo.Setup(p => p.Delete(It.IsAny<Product>()))
        .Callback(new Action<Product>(prod =>
        {
        var productToRemove =
        _products.Find(a => a.ProductId == prod.ProductId);
```

```
        if (productToRemove != null)
        _products.Remove(productToRemove);
        }));

        // Return mock implementation object
        return mockRepo.Object;
}
```

Here we mock all the required methods of Product Repository to get the desired data from _products object and not from actual database.
The single line of code `var mockRepo = new Mock<GenericRepository<Product>>(MockBehavior.Default, _dbEntities);`
mocks the Generic Repository for Product and mockRepo.Setup() mocks the repository methods by passing relevant delegates to the method.

Initialize UnitOfWork and Service

I have written following lines of code in ReInitializeTest() method i.e. our setup method,

```
var unitOfWork = new Mock<IUnitOfWork>();
unitOfWork.SetupGet(s => s.ProductRepository).Returns(_productRepository);
_unitOfWork = unitOfWork.Object;
_productService = new ProductServices(_unitOfWork);
```

Here you can see that I am trying to mock the UnitOfWork instance and forcing it to perform all its transactions and operations on _productRepository that we have mocked earlier. This means that all the transactions will be limited to the mocked repository and actual database or actual repository will not be touched. Same goes for service as well; we are initializing product Services with this mocked _unitOfWork. So when we use _productService in actual tests, it actually works on mocked UnitOfWork and test data only.

All set now and we are ready to write unit tests for ProductService. We'll write test to perform all the CRUD operations that are part of ProductService.

1. GetAllProductsTest ()

Our ProductService in BusinessServices project contains a method named GetAllProducts (), following is the implementation,

```
public IEnumerable<BusinessEntities.ProductEntity> GetAllProducts()
{
	var products = _unitOfWork.ProductRepository.GetAll().ToList();
	if (products.Any())
	{
	Mapper.CreateMap<Product, ProductEntity>();
	var productsModel = Mapper.Map<List<Product>, List<ProductEntity>>(products);
	return productsModel;
	}
	return null;
}
```

We see here, that this method fetches all the available products from the database, maps the database entity to our custom BusinessEntities.ProductEntity and returns the list of custom BusinessEntities.ProductEntity. It returns null if no products are found.

To start writing a test method, you need to decorate that test method with `[Test]` attribute of NUnit framework. This attribute specifies that particular method is a Unit Test method.
Following is the unit test method I have written for the above mentioned business service method,

```
[Test]
public void GetAllProductsTest()
{
	var products = _productService.GetAllProducts();
	var productList =
	products.Select(
	productEntity =>
	new Product {ProductId = productEntity.ProductId, ProductName =
	productEntity.ProductName}).ToList();
	var comparer = new ProductComparer();
	CollectionAssert.AreEqual(
	productList.OrderBy(product => product, comparer),
	_products.OrderBy(product => product, comparer), comparer);
}
```

We used instance of _productService and called the GetAllProducts() method, that will ultimately execute on mocked UnitOfWork and Repository to fetch test data from _products list. The products returned from the method are of type BusinessEntities.ProductEntity and we need to compare the returned products with our

existing _products list i.e. the list of DataModel.Product i.e. a mocked database entity, so we need to convert the returned BusinessEntities.ProductEntity list to DataModel.Product list. We do this with the following line of code,

```
var productList =
products.Select(
productEntity =>
new Product {ProductId = productEntity.ProductId, ProductName =
productEntity.ProductName}).ToList();
```

Now we got two lists to compare, one _products list i.e. the actual products and another productList i.e. the products returned from the service. I have written a helper class and compare method to convert the two Product list in TestHelper project. This method checks the list items and compares them for equality of values. You can add a class named `ProductComparer` to TestHelper project with following implementations,

```
public class ProductComparer : IComparer, IComparer<Product>
{
	public int Compare(object expected, object actual)
	{
		var lhs = expected as Product;
		var rhs = actual as Product;
		if (lhs == null || rhs == null) throw new InvalidOperationException();
		return Compare(lhs, rhs);
	}

	public int Compare(Product expected, Product actual)
	{
		int temp;
		return (temp = expected.ProductId.CompareTo(actual.ProductId)) != 0 ? temp :
		expected.ProductName.CompareTo(actual.ProductName);
	}
}
```

To assert the result we use `CollectionAssert.AreEqual of NUnit where we pass both the lists and comparer.`

```
CollectionAssert.AreEqual(
productList.OrderBy(product => product, comparer),
_products.OrderBy(product => product, comparer), comparer);
```

Since I have NUnit plugin in my visual studio provided by Resharper, let me debug the test method to see the actual result of Assert. We'll run all the tests with NUnit UI at the end of the article.

```
[Test]
public void GetAllProductsTest()
{
    var products = _productService.GetAllProducts();
    var productList =
        products.Select(
            productEntity =>
            new Product {ProductId = productEntity.ProductId, Product
    var comparer = new ProductComparer();
    CollectionAssert.AreEqual(
        productList.OrderBy(product => product, comparer),
        _products.OrderBy(product => product, comparer), comparer);
}
```


productList ,

```
[Test]
public void GetAllProductsTest()
{
    var products = _productService.GetAllProdu
    var productList =
        product
            proc
            new                                    Id = productEr
    var comparer                                   mparer();
    CollectionAssert.AreEqual(
        productList.OrderBy(product => product
        _products.OrderBy(product => product,
}
```

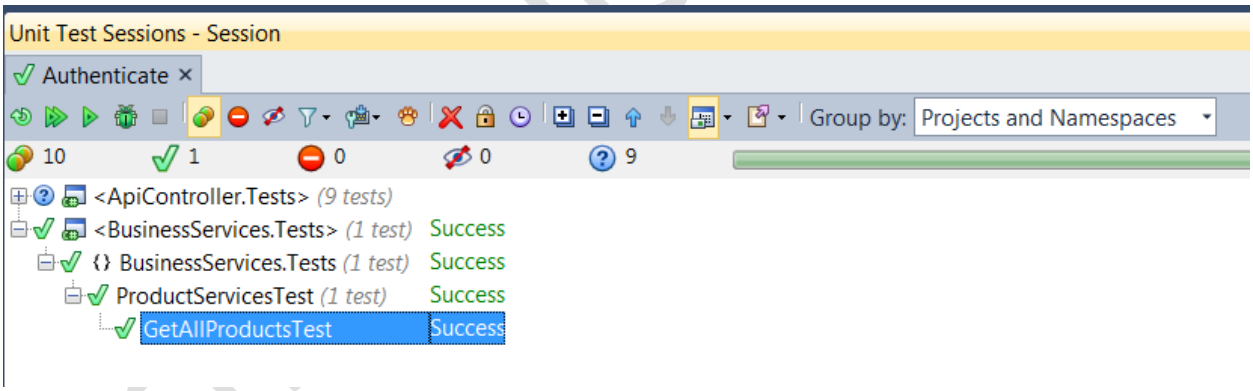

_products,

```
[Test]
public void GetAllProductsTest()
{
    var products = _productService.Ge
    var productList =
        products.Select(
            productEntity =>
            new Product {ProductId =
    var comparer = new ProductCompare
    CollectionAssert.AreEqual(
        productList.OrderBy(product =
        _products.OrderBy(product =>
}

/// <summary>
/// Service shou                    t if
/// </summary>
```

_products Count = 5

[0]	{DataModel.Product}
[1]	{DataModel.Product}
[2]	{DataModel.Product}
[3]	{DataModel.Product}
[4]	{DataModel.Product}
Raw View	

We got both the list, and we need to check the comparison of the lists, I just pressed F5 and got the result on TestUI as,



This shows our test is passed, i.e. the expected and returned result is same.

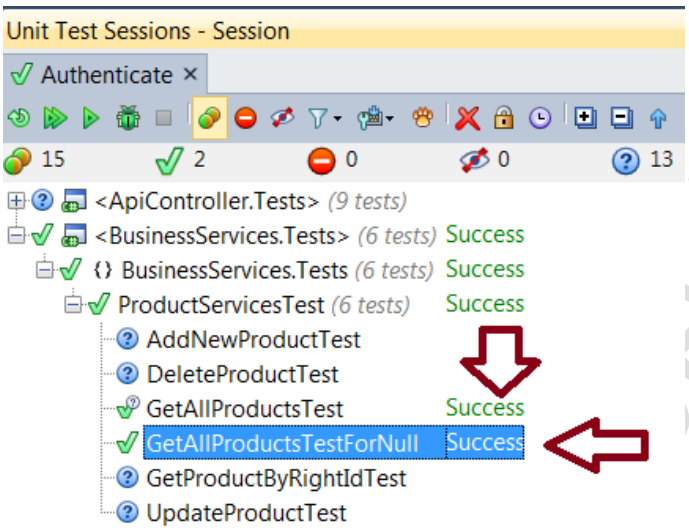
2. GetAllProductsTestForNull ()

You can also write the test for null check for the same method where you nullify the _products list before you invoke the service method. We actually need to write tests that cover all the exit points of the invoked method.

Following test covers another exit point of the method that returns null in case of no products found.

```
/// <summary>
/// Service should return null
/// </summary>
[Test]
public void GetAllProductsTestForNull()
{
    _products.Clear();
    var products = _productService.GetAllProducts();
    Assert.Null(products);
    SetUpProducts();
}
```

In above mentioned test, we first clear the _products list and invoke the service method. Now assert the result for null because our expected result and actual result should be null. I called the SetUpProducts() method again to populate the _products list, but you can do this in test setup method as well i.e. ReInitializeTest().



Now let's move to other tests.

3. GetProductByRightIdTest ()

Here we test GetProductById() method of ProductService. Ideal behavior is that if I invoke the method with a valid id, the method should return the valid product. Now let's suppose I know the product id for my product named "Mobile" and I invoke the test using that id, so ideally I should get a product with the product name mobile.

```
/// <summary>
/// Service should return product if correct id is supplied
/// </summary>
[Test]
public void GetProductByRightIdTest()
{
    var mobileProduct = _productService.GetProductById(2);
```

```
            if (mobileProduct != null)
            {
                Mapper.CreateMap<ProductEntity, Product>();
                var productModel = Mapper.Map<ProductEntity, Product>(mobileProduct);
                AssertObjects.PropertyValuesAreEquals(productModel,
                                                      _products.Find(a =>
a.ProductName.Contains("Mobile")));
            }
        }
```

The above code is self-explanatory except the line `AssertObjects.PropertyValuesAreEquals`.

`_productService.GetProductById(2);` line fetches the product with product id 2.

```
Mapper.CreateMap<ProductEntity, Product>();
var productModel = Mapper.Map<ProductEntity, Product>(mobileProduct);
```

Above code maps the returned custom ProductEntity to DataModel.Product

`AssertObjects` is one more class I have added inside TestHelper class. The purpose of this class is to compare the properties of two objects. This is a common generic class applicable for all type of class objects having properties. Its method PropertyValuesAreEquals() checks for equality of the properties.

AssertObjects class:

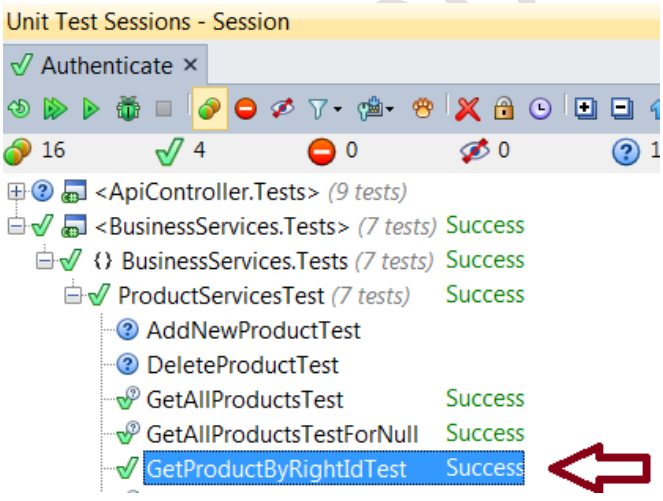
```
using System.Collections;
using System.Reflection;
using NUnit.Framework;

namespace TestsHelper
{
      public static class AssertObjects
      {
            public static void PropertyValuesAreEquals(object actual, object expected)
            {
                  PropertyInfo[] properties = expected.GetType().GetProperties();
                  foreach (PropertyInfo property in properties)
                  {
                        object expectedValue = property.GetValue(expected, null);
                        object actualValue = property.GetValue(actual, null);

                        if (actualValue is IList)
                        AssertListsAreEquals(property, (IList)actualValue,
                        (IList)expectedValue);
                        else if (!Equals(expectedValue, actualValue))
                        if (property.DeclaringType != null)
                        Assert.Fail("Property {0}.{1} does not match. Expected: {2} but was:
                        {3}", property.DeclaringType.Name, property.Name, expectedValue,
                        actualValue);
                  }
            }
```

```
        private static void AssertListsAreEquals(PropertyInfo property, IList actualList,
        IList expectedList)
        {
              if (actualList.Count != expectedList.Count)
              Assert.Fail("Property {0}.{1} does not match. Expected IList containing
              {2} elements but was IList containing {3} elements",
              property.PropertyType.Name, property.Name, expectedList.Count,
              actualList.Count);

              for (int i = 0; i < actualList.Count; i++)
              if (!Equals(actualList[i], expectedList[i]))
              Assert.Fail("Property {0}.{1} does not match. Expected IList with element
              {1} equals to {2} but was IList with element {1} equals to {3}",
              property.PropertyType.Name, property.Name, expectedList[i],
              actualList[i]);
        }
    }
}
```

Running the test,

```
    [Test]
    public void GetProductByRightIdTest()
    {
        var mobileProduct = _productService.GetProductById(2);
        if (mobileProduct != null)
        {
            Mapper.CreateMap<ProductEntity, Product>();
            var productModel = Mapper.Map<ProductEntity, Product>(m
            AssertObjects.PropertyValuesAreEquals(productModel,
                                                  _pr
        }
```

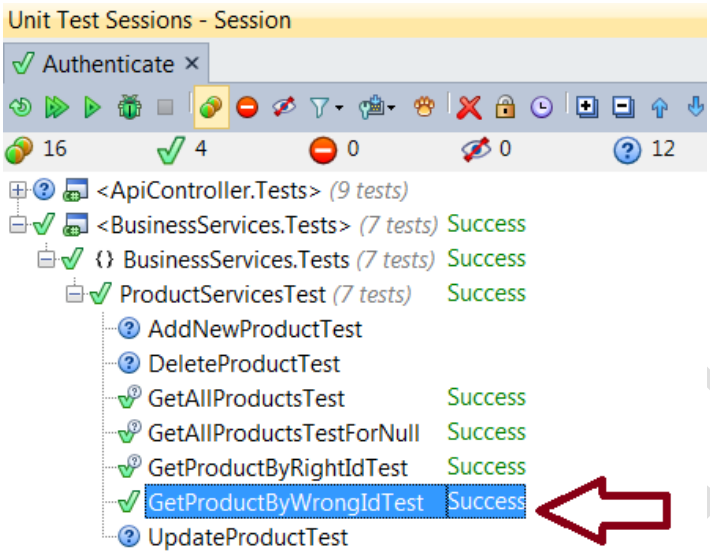
productModel {DataModel.Pr
ProductId 2
ProductName "Mobile"



4. GetProductByWrongIdTest ()

In this test we test the service method with wrong id and expect null in return.

```
/// <summary>
/// Service should return null
/// </summary>
[Test]
public void GetProductByWrongIdTest()
{
    var product = _productService.GetProductById(0);
    Assert.Null(product);
}
```



5. AddNewProductTest ()

In this unit test we test the CreateProduct() method of ProductService. Following is the unit test written for creating a new product.

```
/// <summary>
/// Add new product test
/// </summary>
[Test]
public void AddNewProductTest()
{
      var newProduct = new ProductEntity()
      {
      ProductName = "Android Phone"
      };

      var maxProductIDBeforeAdd = _products.Max(a => a.ProductId);
      newProduct.ProductId = maxProductIDBeforeAdd + 1;
      _productService.CreateProduct(newProduct);
      var addedproduct = new Product() {ProductName = newProduct.ProductName, ProductId =
      newProduct.ProductId};
      AssertObjects.PropertyValuesAreEquals(addedproduct, _products.Last());
```

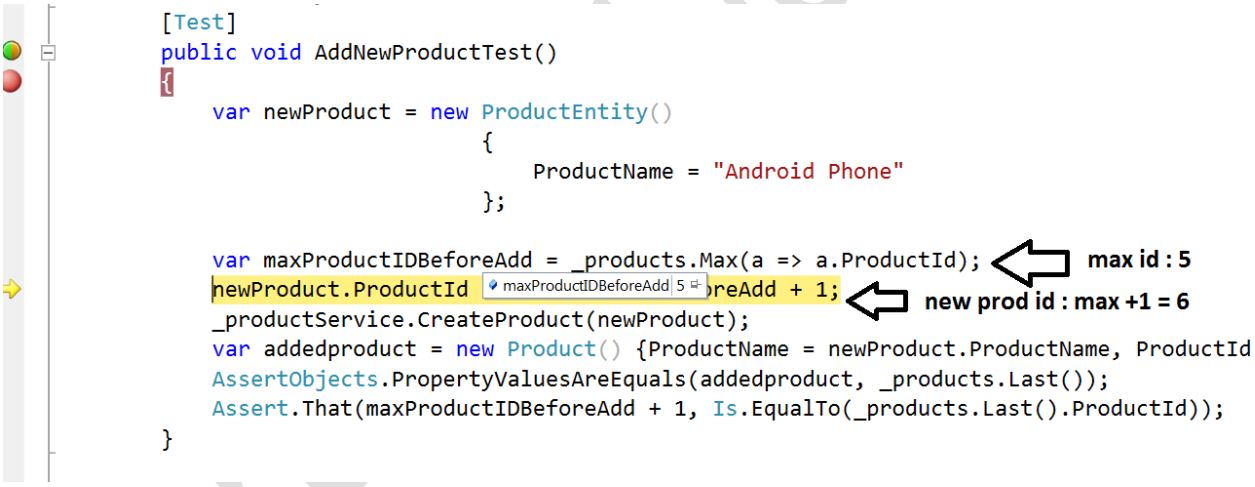
```
        Assert.That(maxProductIDBeforeAdd + 1, Is.EqualTo(_products.Last().ProductId));
}
```

In above code I have created a dummy product with product name “Android Phone” and assigned the product id as the incremented id to the maximum value of productId of the product that lies in _products list. Ideally if my test is success, the added product should reflect in _products list as last product with maximum product id. To verify the result, I have used two asserts, first one checks the properties of expected and actual product and second one verifies the product id.

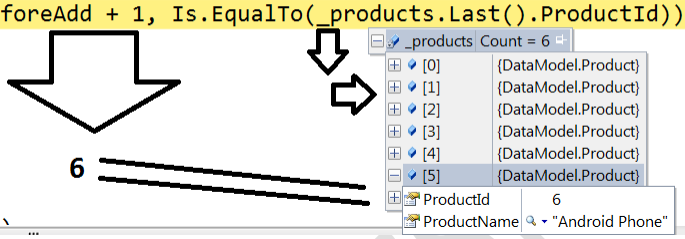
```
var addedproduct = new Product() {ProductName = newProduct.ProductName, ProductId =
newProduct.ProductId};
```

addedProduct is the custom product that is expected to be added in the _products list and _products.Last() gives us last product of the list. So,

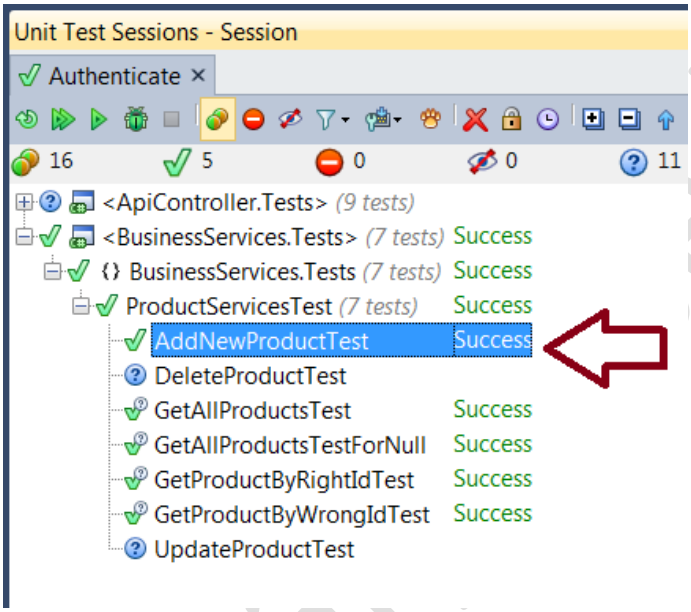
`AssertObjects.PropertyValuesAreEquals(addedproduct, _products.Last());` checks for all the properties of dummy as well as last added product and,
`Assert.That(maxProductIDBeforeAdd + 1, Is.EqualTo(_products.Last().ProductId));` checks if the last product added has the same product id as supplied while creating the product.



```
        [Test]
        public void AddNewProductTest()
        {
            var newProduct = new ProductEntity()
                                {
                                    ProductName = "Android Phone"
                                };

            var maxProductIDBeforeAdd = _products.Max(a => a.ProductId);
            newProduct.ProductId = maxProductIDBeforeAdd + 1;
            _productService.CreateProduct(newProduct);
            var addedproduct = new Product() {ProductName = newProduct.ProductName, Product
            AssertObjects.PropertyValuesAreEquals(addedproduct, _products.Last());
            Assert.That(maxProductIDBeforeAdd + 1, Is.EqualTo(_products.Last().ProductId));
        }

        /// <summary>
        /// Update product test
        /// </summary>
        [Test]
```



After full execution,



The test passes, that means the expected value that was product id 6 was equal to the product id of the last added product in _products list. And we can also see that, earlier we had only 5 products in _products list and now we have added a 6th one.

6. UpdateProductTest ()

This is the unit test to check if the product is updated or not. This test is for UpdateProduct() method of ProductService.

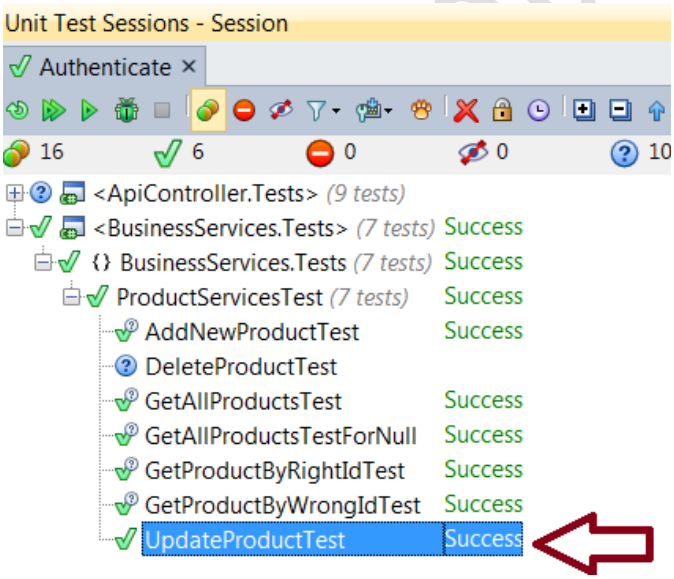
```
/// <summary>
/// Update product test
```

```
/// </summary>
[Test]
public void UpdateProductTest()
{
       var firstProduct = _products.First();
       firstProduct.ProductName = "Laptop updated";
       var updatedProduct = new ProductEntity()
       {ProductName = firstProduct.ProductName, ProductId = firstProduct.ProductId};
       _productService.UpdateProduct(firstProduct.ProductId, updatedProduct);
       Assert.That(firstProduct.ProductId, Is.EqualTo(1)); // hasn't changed
      Assert.That(firstProduct.ProductName, Is.EqualTo("Laptop updated")); // Product name
changed

}
```

In this test I am trying to update first product from _products list. I have changed the product name to “Laptop Updated” and invoked the UpdateProduct () method of ProductService. I have made two asserts to check the updated product from _products list, one for productId and second for product name. We see that we get the updated product while we assert.

```
[Test]
public void UpdateProductTest()
{
    var firstProduct = _products.First();
    firstProduct.ProductName = "Laptop updated";
    var updatedProduct = new ProductEntity()
                             {ProductName = firstProduct.ProductName, Prod
    _productService.UpdateProduct(firstProduct.ProductId, updatedProduct);
    Assert.That(firstProduct.ProductId, Is.EqualTo(1)); // hasn't changed
    Assert.That(firstProduct.ProductName, Is.EqualTo("Laptop updated")); /
}
```

firstProduct.ProductName "Laptop updated"

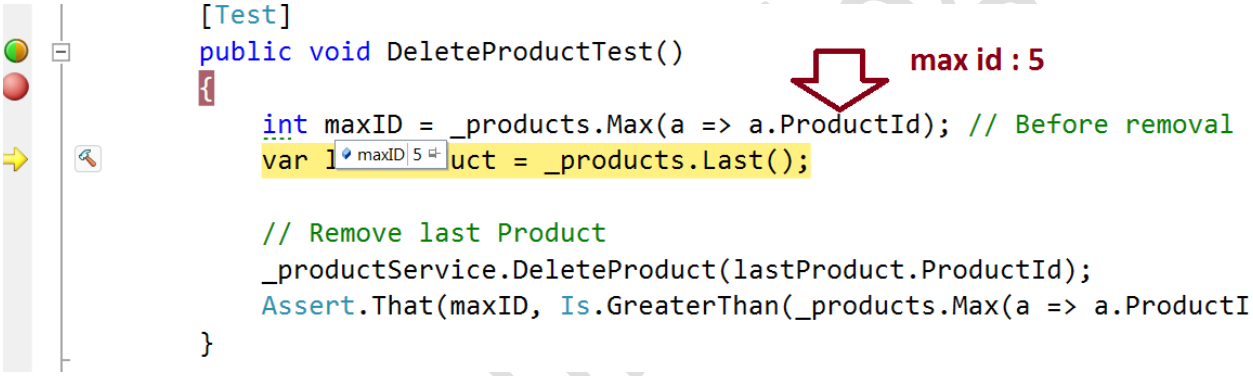


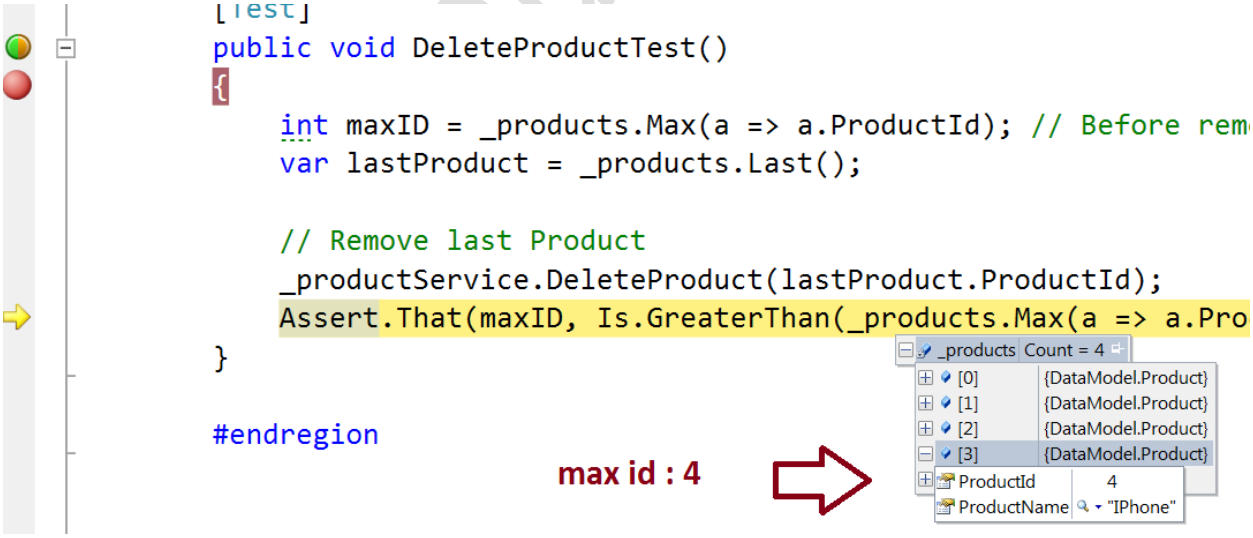
7. DeleteProductTest ()

Following is the test for DeleteProduct () method in ProductService.

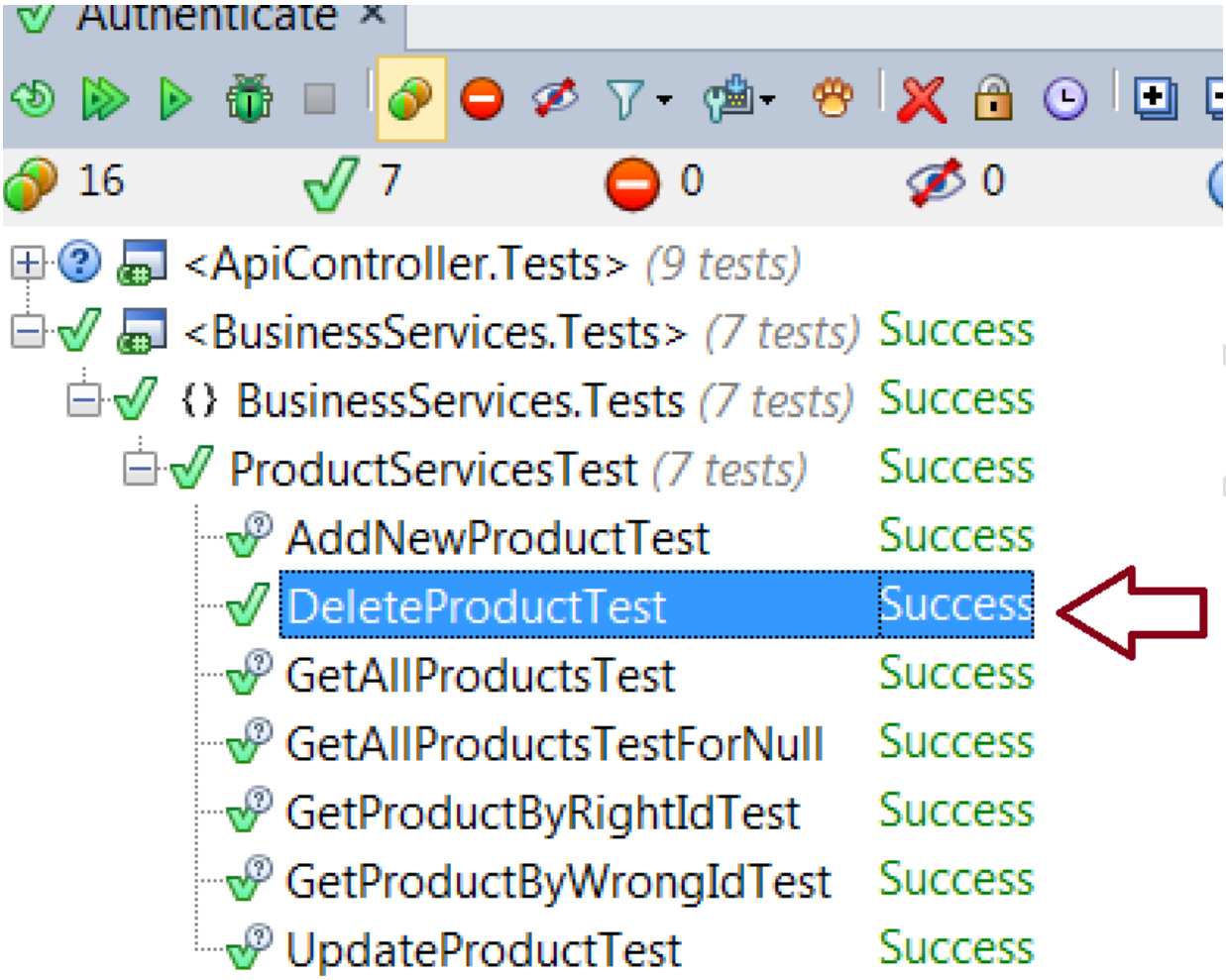
```
        /// <summary>
        /// Delete product test
        /// </summary>
        [Test]
        public void DeleteProductTest()
        {
            int maxID = _products.Max(a => a.ProductId); // Before removal
            var lastProduct = _products.Last();

            // Remove last Product
            _productService.DeleteProduct(lastProduct.ProductId);
            Assert.That(maxID, Is.GreaterThan(_products.Max(a => a.ProductId)));   // Max id
reduced by 1
        }
```

I have written the test to verify the max id of product from the list of products. Get max id of the product, delete the last product and check the max id of the product from the list. The prior max id should be greater than the last product's product id.





Max id before delete was 5 and after delete is 4 that means a product is deleted from _products list therefore statement : `Assert.That(maxID, Is.GreaterThan(_products.Max(a => a.ProductId)));` passes as 5 is greater than 4.



We have covered all the methods of ProductService under unit tests. Following is the final class that covers all the tests for this service.

```
#region using namespaces.
using System;
using System.Collections.Generic;
using System.Linq;
using AutoMapper;
using BusinessEntities;
using DataModel;
using DataModel.GenericRepository;
using DataModel.UnitOfWork;
using Moq;
using NUnit.Framework;
using TestsHelper;

#endregion

namespace BusinessServices.Tests
{
        /// <summary>
        /// Product Service Test
        /// </summary>
```

```
public class ProductServicesTest
{
	#region Variables

	private IProductServices _productService;
	private IUnitOfWork _unitOfWork;
	private List<Product> _products;
	private GenericRepository<Product> _productRepository;
	private WebApiDbEntities _dbEntities;
	#endregion

	#region Test fixture setup

	/// <summary>
	/// Initial setup for tests
	/// </summary>
	[TestFixtureSetUp]
	public void Setup()
	{
	_products = SetUpProducts();
	}

	#endregion

	#region Setup

	/// <summary>
	/// Re-initializes test.
	/// </summary>
	[SetUp]
	public void ReInitializeTest()
	{
	_dbEntities = new Mock<WebApiDbEntities>().Object;
	_productRepository = SetUpProductRepository();
	var unitOfWork = new Mock<IUnitOfWork>();
	unitOfWork.SetupGet(s => s.ProductRepository).Returns(_productRepository);
	_unitOfWork = unitOfWork.Object;
	_productService = new ProductServices(_unitOfWork);
	}

	#endregion

	#region Private member methods

	/// <summary>
	/// Setup dummy repository
	/// </summary>
	/// <returns></returns>
	private GenericRepository<Product> SetUpProductRepository()
	{
	// Initialise repository
	var mockRepo = new Mock<GenericRepository<Product>>(MockBehavior.Default,
	_dbEntities);

	// Setup mocking behavior
	mockRepo.Setup(p => p.GetAll()).Returns(_products);
```

```
mockRepo.Setup(p => p.GetByID(It.IsAny<int>()))
.Returns(new Func<int, Product>(
id => _products.Find(p => p.ProductId.Equals(id))));

mockRepo.Setup(p => p.Insert((It.IsAny<Product>())))
.Callback(new Action<Product>(newProduct =>
{
dynamic maxProductID = _products.Last().ProductId;
dynamic nextProductID = maxProductID + 1;
newProduct.ProductId = nextProductID;
_products.Add(newProduct);
}));

mockRepo.Setup(p => p.Update(It.IsAny<Product>()))
.Callback(new Action<Product>(prod =>
{
var oldProduct = _products.Find(a => a.ProductId == prod.ProductId);
oldProduct = prod;
}));

mockRepo.Setup(p => p.Delete(It.IsAny<Product>()))
.Callback(new Action<Product>(prod =>
{
var productToRemove =
_products.Find(a => a.ProductId == prod.ProductId);

if (productToRemove != null)
_products.Remove(productToRemove);
}));

// Return mock implementation object
return mockRepo.Object;
}

/// <summary>
/// Setup dummy products data
/// </summary>
/// <returns></returns>
private static List<Product> SetUpProducts()
{
var prodId = new int();
var products = DataInitializer.GetAllProducts();
foreach (Product prod in products)
prod.ProductId = ++prodId;
return products;

}

#endregion

#region Unit Tests

/// <summary>
/// Service should return all the products
/// </summary>
```

```
[Test]
public void GetAllProductsTest()
{
var products = _productService.GetAllProducts();
if (products != null)
{
var productList =
products.Select(
productEntity =>
new Product {ProductId = productEntity.ProductId, ProductName =
productEntity.ProductName}).
ToList();
var comparer = new ProductComparer();
CollectionAssert.AreEqual(
productList.OrderBy(product => product, comparer),
_products.OrderBy(product => product, comparer), comparer);
}
}

/// <summary>
/// Service should return null
/// </summary>
[Test]
public void GetAllProductsTestForNull()
{
_products.Clear();
var products = _productService.GetAllProducts();
Assert.Null(products);
SetUpProducts();
}

/// <summary>
/// Service should return product if correct id is supplied
/// </summary>
[Test]
public void GetProductByRightIdTest()
{
var mobileProduct = _productService.GetProductById(2);
if (mobileProduct != null)
{
Mapper.CreateMap<ProductEntity, Product>();
var productModel = Mapper.Map<ProductEntity, Product>(mobileProduct);
AssertObjects.PropertyValuesAreEquals(productModel,
_products.Find(a => a.ProductName.Contains("Mobile")));
}
}

/// <summary>
/// Service should return null
/// </summary>
[Test]
public void GetProductByWrongIdTest()
{
var product = _productService.GetProductById(0);
Assert.Null(product);
}
```

```
/// <summary>
/// Add new product test
/// </summary>
[Test]
public void AddNewProductTest()
{
var newProduct = new ProductEntity()
{
ProductName = "Android Phone"
};

var maxProductIDBeforeAdd = _products.Max(a => a.ProductId);
newProduct.ProductId = maxProductIDBeforeAdd + 1;
_productService.CreateProduct(newProduct);
var addedproduct = new Product() {ProductName = newProduct.ProductName, ProductId
= newProduct.ProductId};
AssertObjects.PropertyValuesAreEquals(addedproduct, _products.Last());
Assert.That(maxProductIDBeforeAdd + 1, Is.EqualTo(_products.Last().ProductId));
}

/// <summary>
/// Update product test
/// </summary>
[Test]
public void UpdateProductTest()
{
var firstProduct = _products.First();
firstProduct.ProductName = "Laptop updated";
var updatedProduct = new ProductEntity()
{ProductName = firstProduct.ProductName, ProductId = firstProduct.ProductId};
_productService.UpdateProduct(firstProduct.ProductId, updatedProduct);
Assert.That(firstProduct.ProductId, Is.EqualTo(1)); // hasn't changed
Assert.That(firstProduct.ProductName, Is.EqualTo("Laptop updated")); // Product
name changed
}

/// <summary>
/// Delete product test
/// </summary>
[Test]
public void DeleteProductTest()
{
int maxID = _products.Max(a => a.ProductId); // Before removal
var lastProduct = _products.Last();

// Remove last Product
_productService.DeleteProduct(lastProduct.ProductId);
Assert.That(maxID, Is.GreaterThan(_products.Max(a => a.ProductId))); // Max id
reduced by 1
}

#endregion


#region Tear Down
```

```
            /// <summary>
            /// Tears down each test data
            /// </summary>
            [TearDown]
            public void DisposeTest()
            {
            _productService = null;
            _unitOfWork = null;
            _productRepository = null;
            if (_dbEntities != null)
            _dbEntities.Dispose();
            }

            #endregion

            #region TestFixture TearDown.

            /// <summary>
            /// TestFixture teardown
            /// </summary>
            [TestFixtureTearDown]
            public void DisposeAllObjects()
            {
            _products = null;
            }

            #endregion
        }
}
```

TokenService Tests

Now that we have completed all the tests for ProductService, I am sure you must have got an idea on how to write unit tests for methods. Note that primarily unit tests are only written to publically exposed methods because the private methods automatically get tested through those public methods in the class. I'll not explain too much theory for TokenService tests and only navigate through code. I'll explain the details wherever necessary.

Tests Setup

Add a new class named TokenServicesTests.cs in BusinessServices.Tests project.

Declare variables

Define the private variable that we'll use in the class to write tests,

```
        #region Variables
        private ITokenServices _tokenServices;
        private IUnitOfWork _unitOfWork;
```

```
private List<Token> _tokens;
private GenericRepository<Token> _tokenRepository;
private WebApiDbEntities _dbEntities;
private const string SampleAuthToken = "9f907bdf-f6de-425d-be5b-b4852eb77761";
#endregion
```

Here _tokenService will hold mock for TokenServices, _unitOfWork for UnitOfWork class, __tokens will hold dummy tokens from DataInitializer class of TestHelper project, _tokenRepository and _dbEntities holds mock for Token Repository and WebAPIDbEntities from DataModel project respectively

Write Test Fixture Setup

```
#region Test fixture setup

/// <summary>
/// Initial setup for tests
/// </summary>
[TestFixtureSetUp]
public void Setup()
{
    _tokens = SetUpTokens();
}

#endregion
```

SetUpTokens () method fetches tokens from DataInitializer class and not from database and assigns a unique id to each token by iterating on them.

```
/// <summary>
/// Setup dummy tokens data
/// </summary>
/// <returns></returns>
private static List<Token> SetUpTokens()
{
    var tokId = new int();
    var tokens = DataInitializer.GetAllTokens();
    foreach (Token tok in tokens)
        tok.TokenId = ++tokId;
    return tokens;
}
```

The result data is assigned to __tokens list to be used in setting up mock repository and in every individual test for comparison of actual vs resultant output.

Write Test Fixture Tear Down

```
#region TestFixture TearDown.

/// <summary>
/// TestFixture teardown
/// </summary>
```

```
        [TestFixtureTearDown]
        public void DisposeAllObjects()
        {
            _tokens = null;
        }

        #endregion
```

Write Test Setup

```
#region Setup

      /// <summary>
      /// Re-initializes test.
      /// </summary>
      [SetUp]
      public void ReInitializeTest()
      {
            _dbEntities = new Mock<WebApiDbEntities>().Object;
            _tokenRepository = SetUpTokenRepository();
            var unitOfWork = new Mock<IUnitOfWork>();
            unitOfWork.SetupGet(s => s.TokenRepository).Returns(_tokenRepository);
            _unitOfWork = unitOfWork.Object;
            _tokenServices = new TokenServices(_unitOfWork);
      }

#endregion
```

Write Test Tear down

```
        #region Tear Down

        /// <summary>
        /// Tears down each test data
        /// </summary>
        [TearDown]
        public void DisposeTest()
        {
            _tokenServices = null;
            _unitOfWork = null;
            _tokenRepository = null;
            if (_dbEntities != null)
                _dbEntities.Dispose();
        }

        #endregion
```

Mocking Repository

```
private GenericRepository<Token> SetUpTokenRepository()
{
// Initialise repository
var mockRepo = new Mock<GenericRepository<Token>>(MockBehavior.Default, _dbEntities);

// Setup mocking behavior
mockRepo.Setup(p => p.GetAll()).Returns(_tokens);

mockRepo.Setup(p => p.GetByID(It.IsAny<int>()))
.Returns(new Func<int, Token>(
id => _tokens.Find(p => p.TokenId.Equals(id))));

mockRepo.Setup(p => p.GetByID(It.IsAny<string>()))
.Returns(new Func<string, Token>(
authToken => _tokens.Find(p => p.AuthToken.Equals(authToken))));

mockRepo.Setup(p => p.Insert((It.IsAny<Token>())))
.Callback(new Action<Token>(newToken =>
{
dynamic maxTokenID = _tokens.Last().TokenId;
dynamic nextTokenID = maxTokenID + 1;
newToken.TokenId = nextTokenID;
_tokens.Add(newToken);
}));

mockRepo.Setup(p => p.Update(It.IsAny<Token>()))
.Callback(new Action<Token>(token =>
{
var oldToken = _tokens.Find(a => a.TokenId == token.TokenId);
oldToken = token;
}));

mockRepo.Setup(p => p.Delete(It.IsAny<Token>()))
.Callback(new Action<Token>(prod =>
{
var tokenToRemove =
_tokens.Find(a => a.TokenId == prod.TokenId);

if (tokenToRemove != null)
_tokens.Remove(tokenToRemove);
}));
//Create setup for other methods too. note non virtauls methods can not be set up

// Return mock implementation object
return mockRepo.Object;
}
```

Note, while mocking repository, I have setup two mocks for GetById(). There is a minor change I did in the database, I have marked AuthToken field as a primary key too. So it may be a situation where mock gets confused on calling the method that for which primary key the request has been made. So I have implemented the mock both for TokenId and AuthToken field,

```
mockRepo.Setup(p => p.GetByID(It.IsAny<int>())).Returns(new Func<int, Token>(
id => _tokens.Find(p => p.TokenId.Equals(id))));

mockRepo.Setup(p => p.GetByID(It.IsAny<string>())).Returns(new Func<string, Token>(
authToken => _tokens.Find(p => p.AuthToken.Equals(authToken))));
```

The overall setup is of same nature as we wrote for ProductService. Let us move on to unit tests.

1. GenerateTokenByUseridTest ()

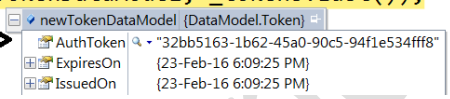
This unit test is to test the GenerateToken method of TokenServices business service. In this method a new token is generated in the database against a user. We'll use _tokens list for all these transactions. Currently we have only two token entries in _tokens list generated from DataInitializer. Now when the test executes it should expect one more token to be added to the list.

```
[Test]
 public void GenerateTokenByUserIdTest()
 {
     const int userId = 1;
     var maxTokenIdBeforeAdd = _tokens.Max(a => a.TokenId);
     var tokenEntity = _tokenServices.GenerateToken(userId);
     var newTokenDataModel = new Token()
                                 {
                                     AuthToken = tokenEntity.AuthToken,
                                     TokenId = maxTokenIdBeforeAdd+1,
                                     ExpiresOn = tokenEntity.ExpiresOn,
                                     IssuedOn = tokenEntity.IssuedOn,
                                     UserId = tokenEntity.UserId
                                 };
     AssertObjects.PropertyValuesAreEquals(newTokenDataModel, _tokens.Last());
 }
```

I have taken a default user id as 1, and stored the max token id from the list of tokens. Call the service method GenerateTokenEntity (). Since our service method returns BusinessEntities.TokenEntity, we need to map it to new DataModel.Token object for comparison. So expected result is that all the properties of this token should match the last token of the _token list assuming that list is updated through the test.

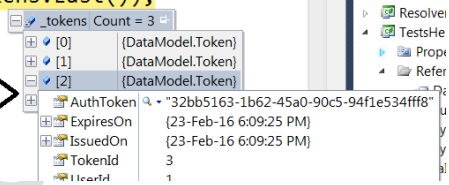
```
        [Test]
        public void GenerateTokenByUserIdTest()
        {
            const int userId = 1;
            var maxTokenIdBeforeAdd = _tokens.Max(a => a.TokenId);
            var tokenEntity = _tokenServices.GenerateToken(userId);
            var newTokenDataModel = new Token()
                                        {
                                            AuthToken = tokenEntity.AuthToken,
                                            TokenId = maxTokenIdBeforeAdd+1,
                                            ExpiresOn = tokenEntity.ExpiresOn,
                                            IssuedOn = tokenEntity.IssuedOn,
                                            UserId = tokenEntity.UserId
                                        };
            AssertObjects.PropertyValuesAreEquals(newTokenDataModel, _tokens.Last());
        }
```

Generated Auth Token



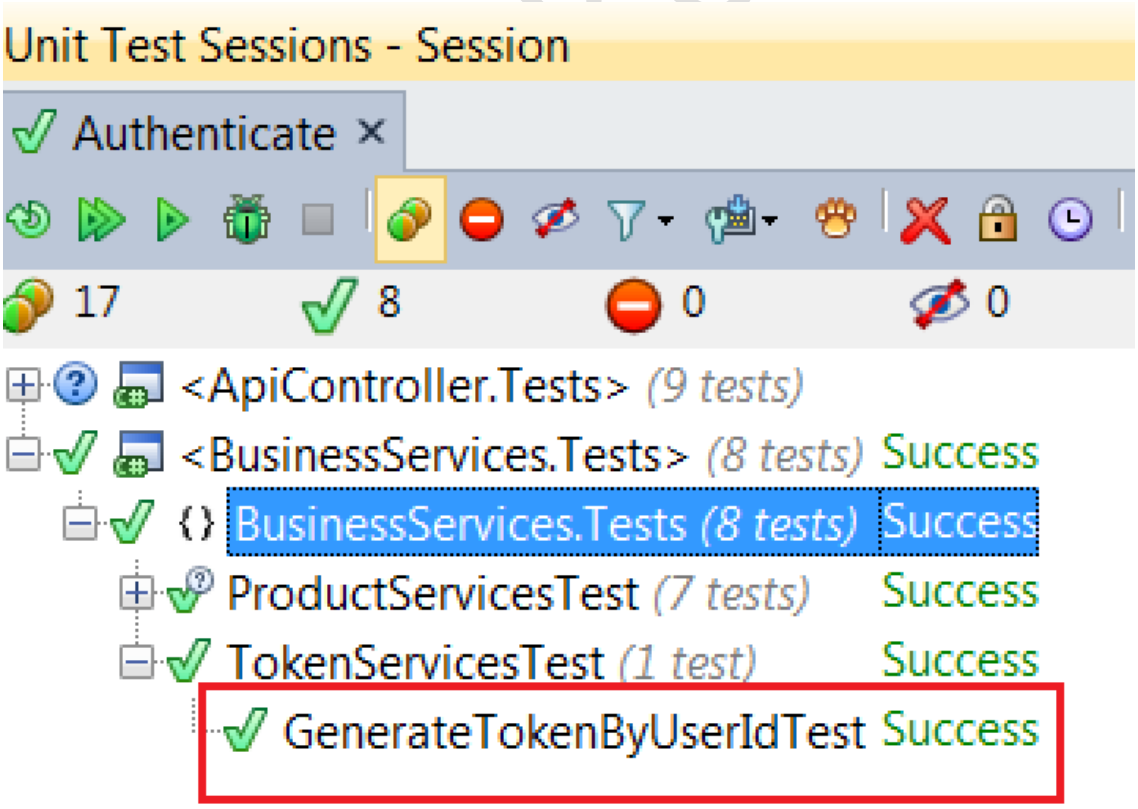

```
                                    };
    AssertObjects.PropertyValuesAreEquals(newTokenDataModel, _tokens.Last());
```

Authtoken for last entry




```
/ <summary>
/ Generate token test
/ </summary>
```

Now since all the properties of the resultant and actual object match, so our test passes.



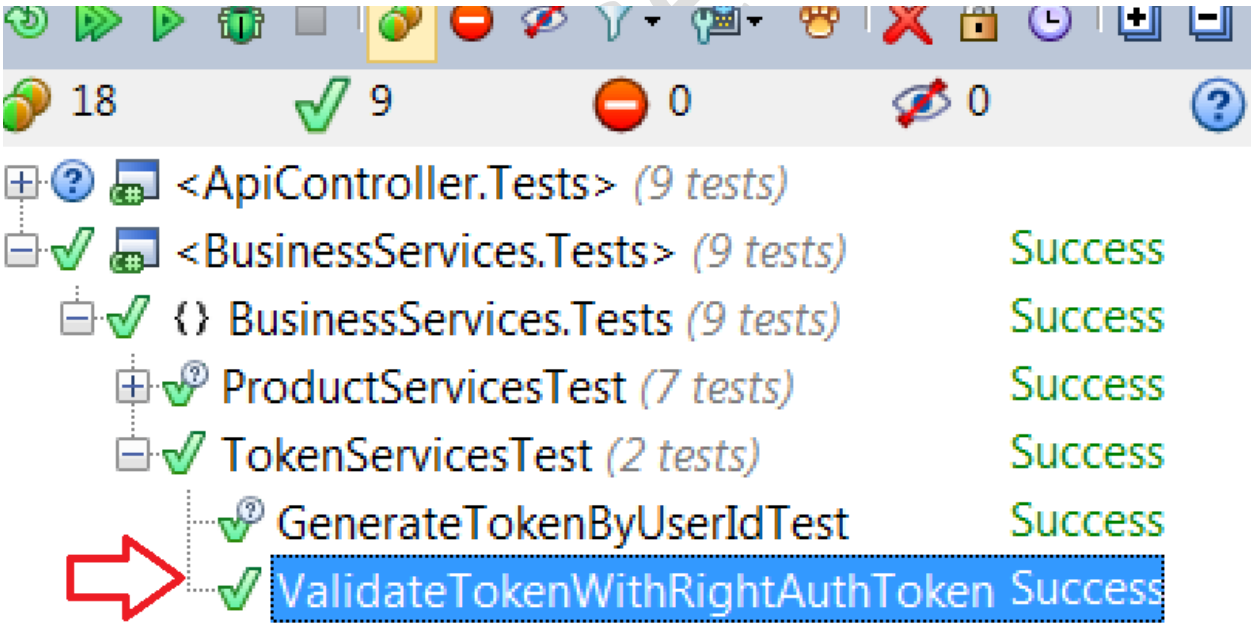
2. ValidateTokenWithRightAuthToken ()

```
/// <summary>
/// Validate token test
/// </summary>
[Test]
public void ValidateTokenWithRightAuthToken()
{
    var authToken = Convert.ToString(SampleAuthToken);
    var validationResult = _tokenServices.ValidateToken(authToken);
    Assert.That(validationResult,Is.EqualTo(true));
}
```

This test validates AuthToken through ValidateToken method of TokenService. Ideally if correct token is passed, the service should return true.

```
[Test]
public void ValidateTokenWithRightAuthToken()
{
    var authToken = Convert.ToString(SampleAuthToken);
    var validationResult = _tokenServices.ValidateToken(authToken);
    Assert.That(validationResult, true Is.EqualTo(true));
}
```

Here we get validationResult as true therefore test should pass.



3. ValidateTokenWithWrongAuthToken ()

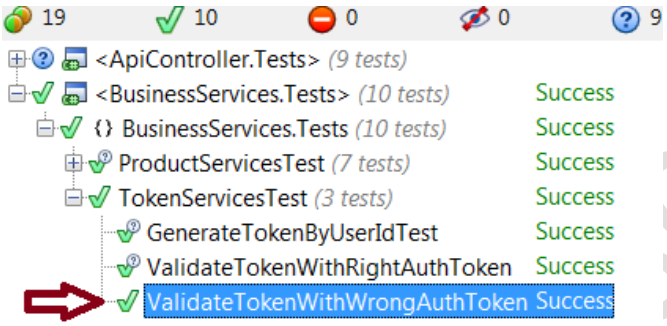
Testing same method for its alternate exit point, therefore with wrong token, the service should return false.

```
        [Test]
         public void ValidateTokenWithWrongAuthToken()
         {
             var authToken = Convert.ToString("xyz");
             var validationResult = _tokenServices.ValidateToken(authToken);
             Assert.That(validationResult, Is.EqualTo(false));
         }
```

```
[Test]
public void ValidateTokenWithWrongAuthToken()
{
    var authToken = Convert.ToString("xyz");
    var validationResult = _tokenServices.ValidateToken(
    Assert.That(validationResult, Is.EqualTo(false));
}
```

validationResult false

Here validationResult is false, and is compared to false value, so test should ideally pass.



UserService Tests

I have tried writing unit tests for UserService as per our service implementations, but encountered an error while mocking up our repositories Get () method that takes predicate or where condition as a parameter.

```
public TEntity Get(Func<TEntity, Boolean> where)
    {
        return DbSet.Where(where).FirstOrDefault<TEntity>();
    }
```

Since our service methods heavily depend on the Get method, so none of the methods could be tested, but apart from this you can search for any other mocking framework that takes care of these situations.I guess this is a bug in mocking framework. Alternatively refrain yourself from using Get method with predicate (I would not suggest this approach as it is against the testing strategy.Our tests should not be limited to technical feasibility of methods). I got following error while mocking repository,

```
mockRepo.Setup(s => s.Get(It.IsAny<Func<User, bool>>()))
    .Returns(
        (Func<User, bool> expr) =>
        DataInitializer.GetAllUsers().Where(u => u.UserName == CorrectUserName).FirstOrDefault(
            u => u.Password == CorrectPassword));

mockRepo.Setup(s => s.Get(It.IsAny<Func<User, bool
    .Returns(
```

NotSupportedException was unhandled by user code

Invalid setup on a non-virtual (overridable in VB) member: s => s.Get(It.IsAny<Func`2>())

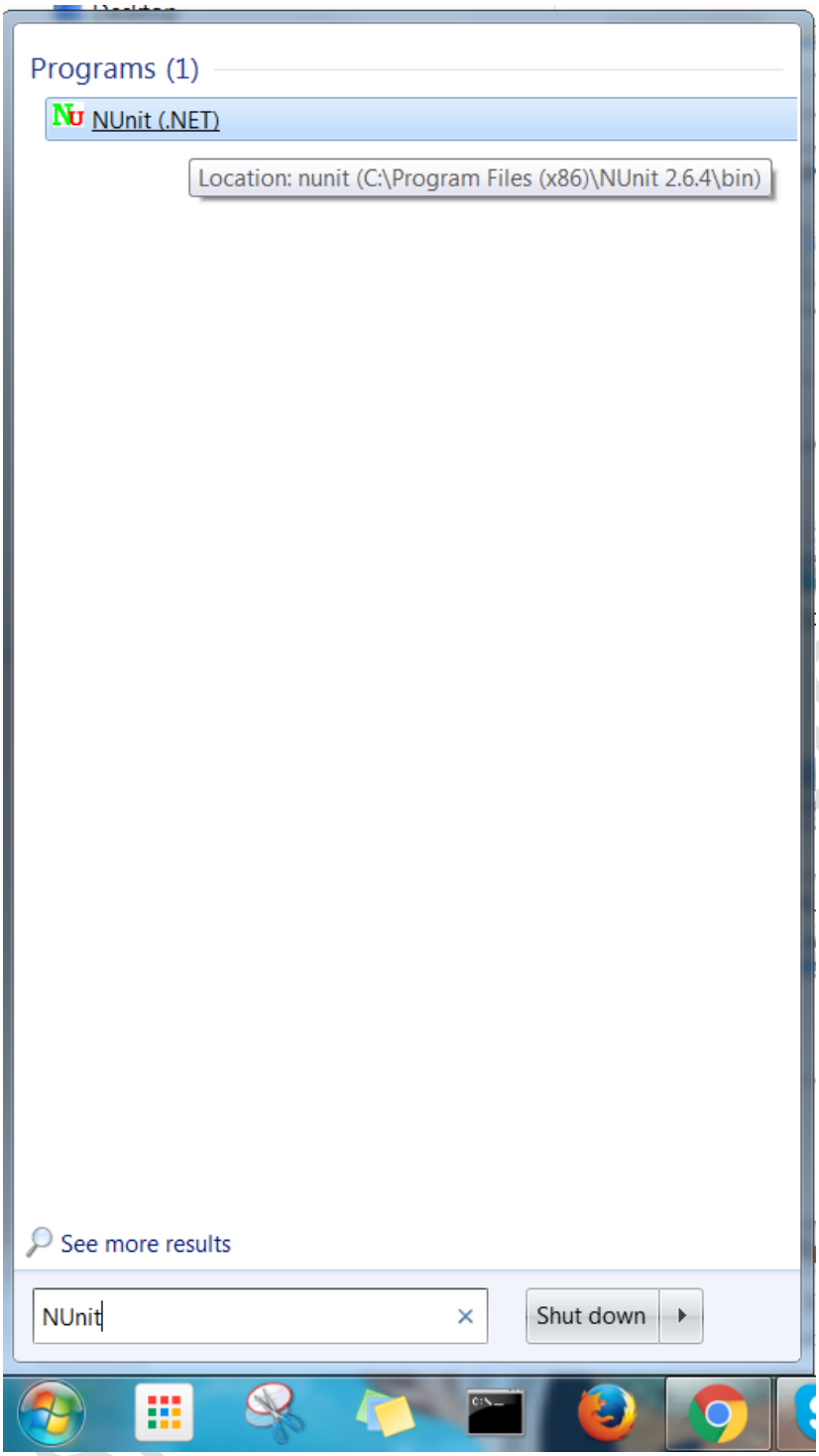
"Invalid setup on a non-virtual (overridable in VB)". I have commented out all UserService Unit test code, you can find it in available source code.

Test through NUnit UI

We have completed almost all the BusinessServices test, now let us try to execute these test on NUnit UI.

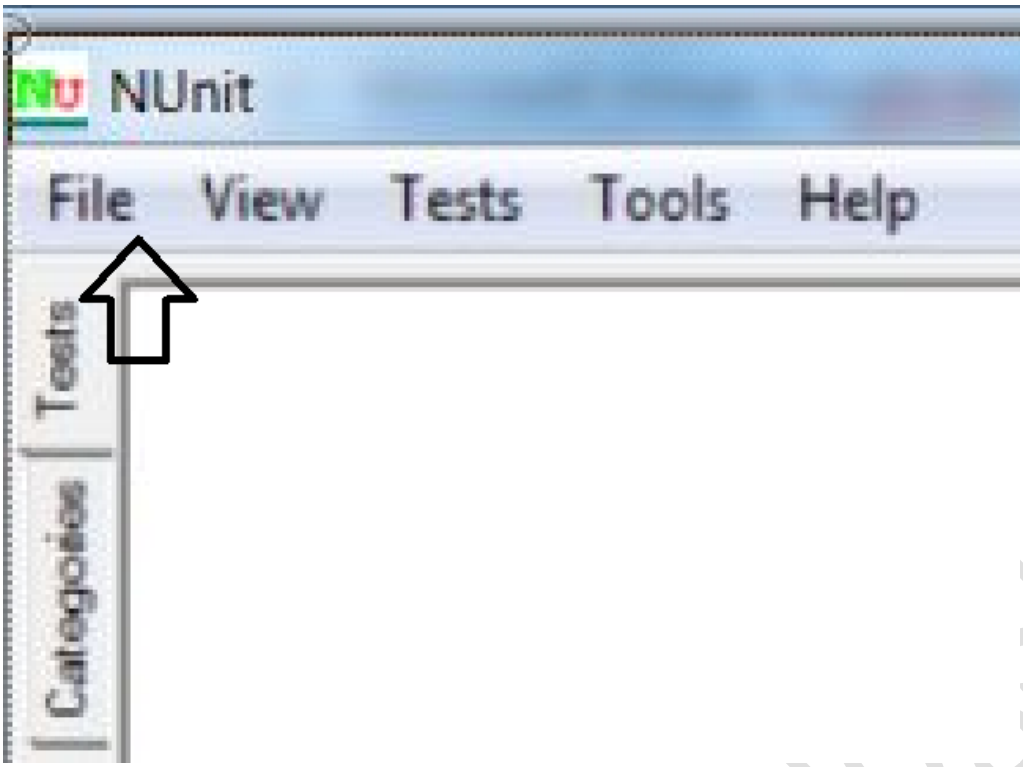
1. Step 1:

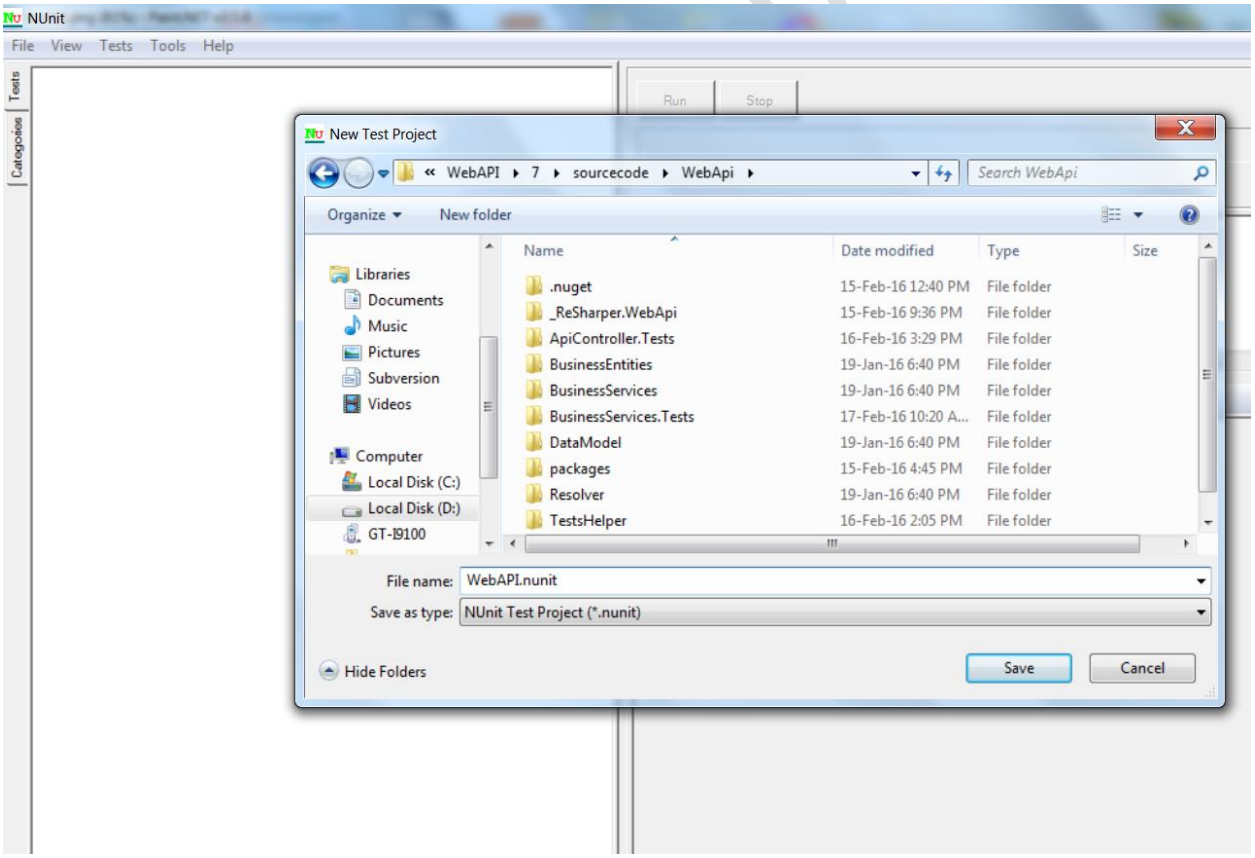
   Launch NUnit UI. I have already explained how to install NUnit on the windows machine. Just launch the NUnit interface with its launch icon,

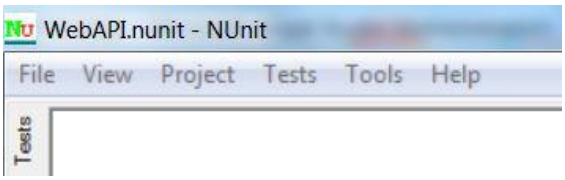


2. Step 2 :

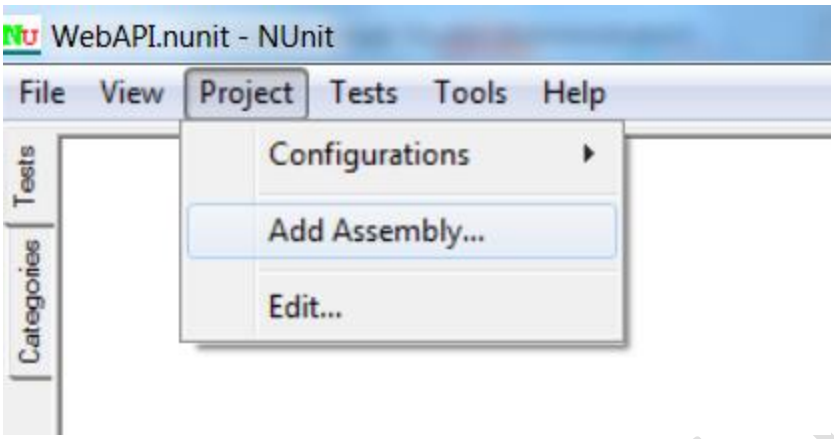
Once the interface opens, click on File -> New Project and name the project as WebAPI.nunit and save it at any windows location.

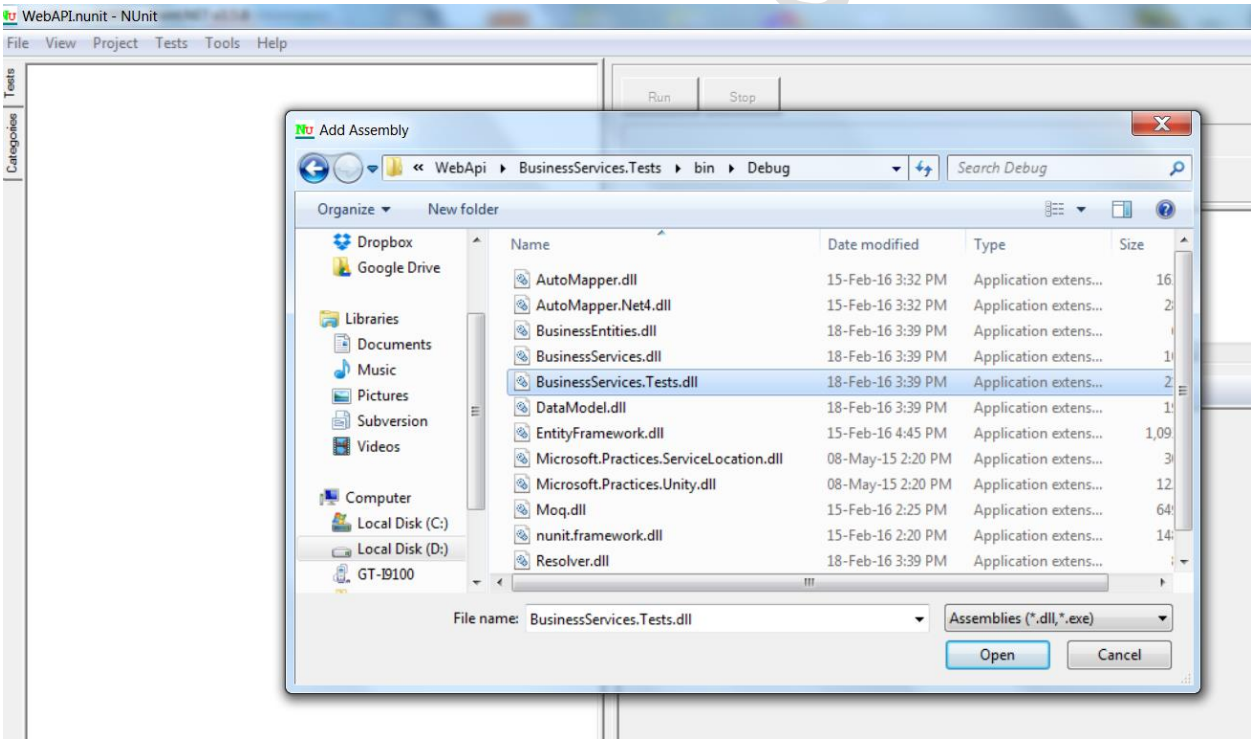

NUnit
File View Tests Tools Help
Tests
Categories


NUnit
File View Tests Tools Help
Tests
Categories
Run
Stop
New Test Project
« WebAPI ▸ 7 ▸ sourcecode ▸ WebApi ▸
Search WebApi
Organize
New folder
Libraries
Documents
Music
Pictures
Subversion
Videos
Computer
Local Disk (C:)
Local Disk (D:)
GT-I9100
Name
Date modified
Type
Size
.nuget 15-Feb-16 12:40 PM File folder
_ReSharper.WebApi 15-Feb-16 9:36 PM File folder
ApiController.Tests 16-Feb-16 3:29 PM File folder
BusinessEntities 19-Jan-16 6:40 PM File folder
BusinessServices 19-Jan-16 6:40 PM File folder
BusinessServices.Tests 17-Feb-16 10:20 A... File folder
DataModel 19-Jan-16 6:40 PM File folder
packages 15-Feb-16 4:45 PM File folder
Resolver 19-Jan-16 6:40 PM File folder
TestsHelper 16-Feb-16 2:05 PM File folder
File name: WebAPI.nunit
Save as type: NUnit Test Project (*.nunit)
Hide Folders
Save
Cancel

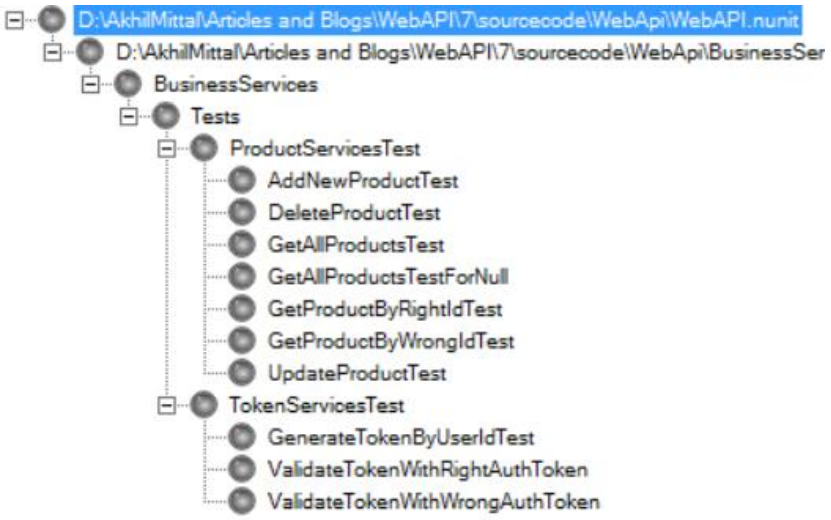


3. Step 3: Now, click on Project-> Add Assembly and browse for BusinessServices.Tests.dll (The library created for your unit test project when compiled)

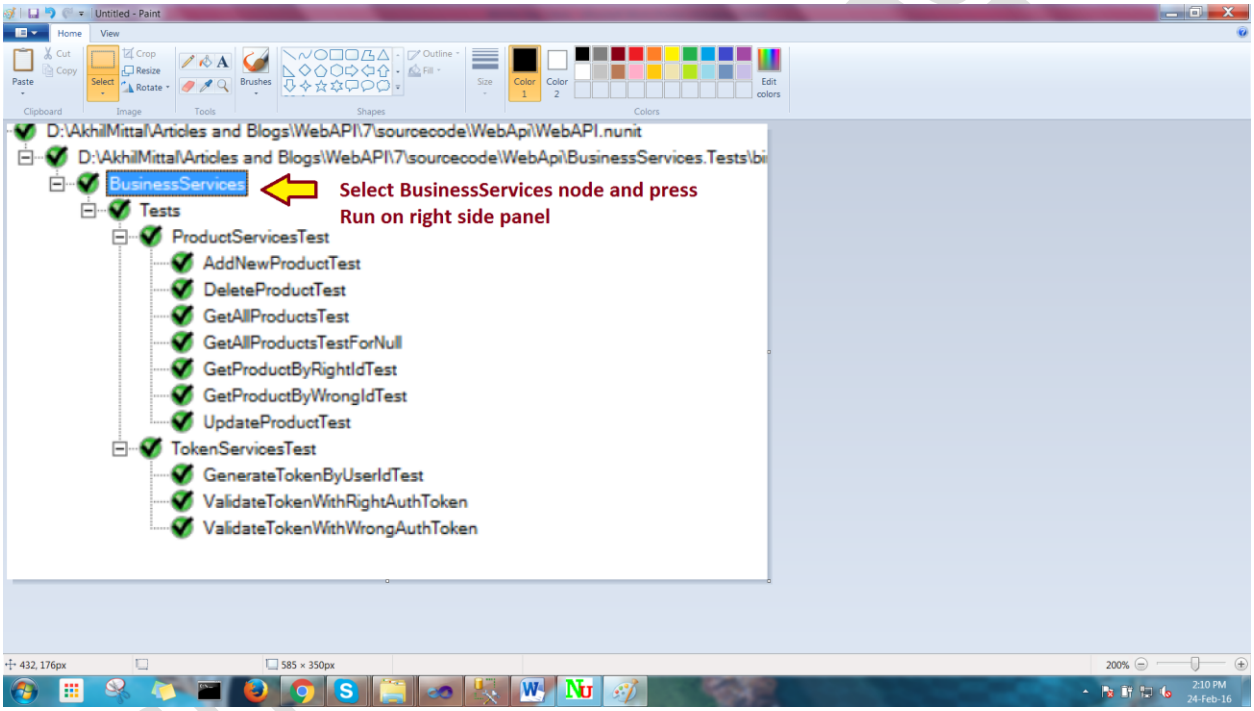




4. Step 4: Once the assembly is browsed, you'll see all the unit tests for that test project gets loaded in the UI and are visible on the interface.



5. At the right hand side panel of the interface, you'll see a Run button that runs all the tests of business service. Just select the node BusinessServices in the tests tree on left side and press Run button on the right side.



Once you run the tests, you'll get green progress bar on right side and tick mark on all the tests on left side. That means all the tests are passed. In case any test fails, you'll get cross mark on the test and red progress bar on right side.

But here, all of our tests are passed.

WebAPI Tests

Unit tests for WebAPI are not exactly like service methods, but vary in terms of testing HttpResponse, returned JSON, exception responses etc. Note that we'll mock the classes and repository in a similar way in WebAPI unit tests as we did for services. One way of testing a web api is through web client and testing the actual endpoint or hosted URL of the service, but that is not considered as a unit test, that is called integration testing. In the next part of the article I'll explain step by step procedure to unit test a web API. We'll write tests for Product Controller.

Conclusion

In this article we learnt how to write unit tests for core business logic and primarily on basic CURD operations. The purpose was to get a basic idea on how unit tests are written and executed. You can add your own flavor to this that helps you in your real time project. My next article which explains unit tests for WebAPI controllers will be the continuation of this part. I hope this was useful to you. You can download the complete source code of this article with packages from **GitHub**. Happy coding ☺

Other Series

My other series of articles:

- Introduction to MVC Architecture and Separation of Concerns: Part 1
- Diving Into OOP (Day 1): Polymorphism and Inheritance (Early Binding/Compile Time Polymorphism)

For more informative articles visit my Blog.