

Diving into ASP.NET WebAPI

Akhil Mittal

Diving into ASP.NET WebAPI

(A Practical Approach of Creating RESTful Services in .NET)

Akhil Mittal

Sr. Analyst (Magic Software)

<https://www.facebook.com/csharppulse>



<https://in.linkedin.com/in/akhilmittal>



<https://twitter.com/AkhilMittal20>



[google.com/+AkhilMittal](https://plus.google.com/+AkhilMittal)



©2016 Akhil Mittal (www.codet Teddy.com)

<https://www.codet Teddy.com>



<https://github.com/akhilmittal>



SHARE THIS DOCUMENT AS IT IS. PLEASE DO NOT REPRODUCE, REPUBLISH, CHANGE OR COPY.

Contents

CONTENTS	4
ABOUT AUTHOR	12
ENTERPRISE-LEVEL APPLICATION ARCHITECTURE WITH WEB APIS USING ENTITY FRAMEWORK, GENERIC REPOSITORY PATTERN AND UNIT OF WORK	14
REST	15
SETUP DATABASE	15
WEB API PROJECT	16
<i>Step 1</i>	17
<i>Step 2</i>	17
<i>Step 3</i>	18
SETUP DATA ACCESS LAYER	19
<i>Step 1</i>	20
<i>Step 2</i>	21
<i>Step 3</i>	21
<i>Step 4</i>	25
<i>Step 5</i>	27
<i>Step 6</i>	27
<i>Step 7</i>	29
GENERIC REPOSITORY AND UNIT OF WORK	30
<i>Step 1</i>	30
<i>Step 2</i>	36
SETUP BUSINESS ENTITIES	41
<i>Product Entity</i>	42
<i>Token entity</i>	42
<i>User Entity</i>	42

SETUP BUSINESS SERVICES PROJECT	43
<i>Step 1</i>	43
<i>Step 2</i>	44
<i>Step 3</i>	44
<i>Step 4</i>	45
SETUP WEBAPI PROJECT	55
<i>Step 1</i>	55
<i>Step 2</i>	56
<i>Step 3</i>	58
RUNNING THE APPLICATION	63
DESIGN FLAWS	66
CONCLUSION	67
SOURCE CODE	67
INVERSION OF CONTROL USING DEPENDENCY INJECTION IN WEB API'S USING UNITY CONTAINER AND BOOTSTRAPPER	68
EXISTING DESIGN AND PROBLEM	69
INTRODUCTION TO UNITY	71
SETUP UNITY	72
<i>Step 1</i>	72
<i>Step 2</i>	73
<i>Step 3</i>	75
<i>Step 4</i>	76
SETUP CONTROLLER	78
SETUP SERVICES	78
RUNNING THE APPLICATION	79
DESIGN FLAWS	88
CONCLUSION	89
SOURCE CODE	89

RESOLVE DEPENDENCY OF DEPENDENCIES USING INVERSION OF CONTROL & DEPENDENCY INJECTION IN ASP.NET WEB APIS WITH UNITY CONTAINER AND MANAGED EXTENSIBILITY FRAMEWORK (MEF)	90
EXISTING DESIGN AND PROBLEM	91
MANAGED EXTENSIBILITY FRAMEWORK (MEF)	93
CREATING A DEPENDENCY RESOLVER WITH UNITY AND MEF	93
SETUP BUSINESS SERVICES	103
SETUP DATAMODEL	105
SETUP REST ENDPOINT / WEBAPI PROJECT	107
RUNNING THE APPLICATION	113
ADVANTAGES OF THIS DESIGN	119
CONCLUSION	120
CUSTOM URL RE-WRITING/ROUTING USING ATTRIBUTE ROUTES IN MVC 4 WEB APIS	121
ROUTING	122
EXISTING DESIGN AND PROBLEM	122
ATTRIBUTE ROUTING	129
SETUP REST ENDPOINT / WEBAPI PROJECT TO DEFINE ROUTES	132
MORE ROUTING CONSTRAINTS	141
RANGE:	141
REGULAR EXPRESSION:	142
OPTIONAL PARAMETERS AND DEFAULT PARAMETERS:	142
ROUTEPREFIX: ROUTING AT CONTROLLER LEVEL	148
ROUTEPREFIX: VERSIONING	150
ROUTEPREFIX: OVERRIDING	151
DISABLE DEFAULT ROUTES	152
RUNNING THE APPLICATION	152
CONCLUSION	158
REFERENCES	158
SECURITY IN WEB API - BASIC & TOKEN BASED CUSTOM AUTHORIZATION IN WEB APIS USING ACTION FILTERS	160

SECURITY IN WEBAPI	161
AUTHENTICATION	162
AUTHORIZATION	162
MAINTAINING SESSION	162
BASIC AUTHENTICATION	162
PROS AND CONS OF BASIC AUTHENTICATION	163
TOKEN BASED AUTHORIZATION	163
WEBAPI WITH BASIC AUTHENTICATION AND TOKEN BASED AUTHORIZATION	164
CREATING USER SERVICE	164
<i>UserServices:</i>	165
<i>Resolve dependency of UserService:</i>	169
IMPLEMENTING BASIC AUTHENTICATION	170
<i>Step 1: Create generic Authentication Filter</i>	170
<i>Step 2: Create Basic Authentication Identity</i>	177
<i>Step 3: Create a Custom Authentication Filter</i>	179
<i>Step 4: Basic Authentication on Controller</i>	183
RUNNING THE APPLICATION	188
DESIGN DISCREPANCY	196
IMPLEMENTING TOKEN BASED AUTHORIZATION	196
<i>Set Database</i>	197
<i>Set Business Services</i>	198
<i>Setup WebAPI/Controller:</i>	209
<i>AuthenticateController:</i>	209
<i>Setup Authorization Action Filter</i>	217
<i>Mark Controllers with Authorization filter</i>	220
MAINTAINING SESSION USING TOKEN	221
RUNNING THE APPLICATION	223
TEST AUTHENTICATION	223

TEST AUTHORIZATION	225
CONCLUSION	229
REFERENCES	230
REQUEST LOGGING AND EXCEPTION HANDLING/LOGGING IN WEB APIS USING ACTION FILTERS, EXCEPTION FILTERS AND NLOG	231
REQUEST LOGGING	232
SETUP NLOG IN WEBAPI	232
<i>Step 1: Download NLog Package</i>	233
<i>Step 2: Configuring NLog</i>	235
NLOGGER CLASS	236
ADDING ACTION FILTER	242
<i>Step 1: Adding LoggingFilterAttribute class</i>	242
<i>Step 2: Registering Action Filter (LoggingFilterAttribute)</i>	244
RUNNING THE APPLICATION	245
EXCEPTION LOGGING	254
IMPLEMENTING EXCEPTION LOGGING	254
<i>Step 1: Exception Filter Attribute</i>	254
<i>Step 2: Modify NLogger Class</i>	257
<i>Step 3: Modify Controller for Exceptions</i>	259
<i>Step 4: Run the application</i>	260
CUSTOM EXCEPTION LOGGING	264
JSON SERIALIZERS	274
MODIFY NLOGGER CLASS	276
MODIFY GLOBALEXCEPTIONATTRIBUTE	280
MODIFY PRODUCT CONTROLLER	282
RUN THE APPLICATION	283
UPDATE THE CONTROLLER FOR NEW EXCEPTION HANDLING	287
<i>Product Controller</i>	288

CONCLUSION	293
UNIT TESTING AND INTEGRATION TESTING IN WEBAPI USING NUNIT AND MOQ FRAMEWORK: PART 1	295
UNIT TESTS	296
NUNIT	297
MOQ FRAMEWORK	299
SETUP SOLUTION	300
TESTING BUSINESS SERVICES	310
STEP 1: TEST PROJECT	310
STEP 2: INSTALL NUNIT PACKAGE	311
STEP 3: INSTALL MOQ FRAMEWORK	312
STEP 4: INSTALL ENTITY FRAMEWORK	313
STEP 5: INSTALL AUTOMAPPER	313
STEP 6: REFERENCES	314
TESTHELPER	314
PRODUCTSERVICE TESTS	319
<i>Tests Setup</i>	319
1. <i>GetAllProductsTest ()</i>	328
2. <i>GetAllProductsTestForNull ()</i>	333
3. <i>GetProductByRightIdTest ()</i>	335
4. <i>GetProductByWrongIdTest ()</i>	339
5. <i>AddNewProductTest ()</i>	340
6. <i>UpdateProductTest ()</i>	343
7. <i>DeleteProductTest ()</i>	345
TOKENSERVICE TESTS	359
<i>Tests Setup</i>	359
1. <i>GenerateTokenByUserIdTest ()</i>	367
2. <i>ValidateTokenWithRightAuthToken ()</i>	369
3. <i>ValidateTokenWithWrongAuthToken ()</i>	371

USERSERVICE TESTS	372
WEBAPI TESTS	378
CONCLUSION	378
UNIT TESTING AND INTEGRATION TESTING IN WEBAPI USING NUNIT AND MOQ FRAMEWORK: PART 2	380
SETUP SOLUTION	381
TESTING WEBAPI	382
STEP 1: TEST PROJECT	382
STEP 2: INSTALL NUNIT PACKAGE	382
STEP 3: INSTALL MOQ FRAMEWORK	383
STEP 4: INSTALL ENTITY FRAMEWORK	383
STEP 5: NEWTONSOFT.JSON	384
STEP 6: REFERENCES	385
PRODUCTCONTROLLER TESTS	385
<i>Tests Setup</i>	385
8. <i>GetAllProductsTest ()</i>	397
9. <i>GetProductByIdTest ()</i>	400
10. <i>GetProductByWrongIdTest ()</i>	403
11. <i>GetProductByInvalidIdTest ()</i>	405
12. <i>CreateProductTest ()</i>	407
13. <i>UpdateProductTest ()</i>	408
14. <i>DeleteProductTest ()</i>	409
15. <i>DeleteInvalidProductTest ()</i>	411
16. <i>DeleteProductWithWrongIdTest ()</i>	412
INTEGRATION TESTS	419
DIFFERENCE BETWEEN UNIT TESTS AND INTEGRATION TESTS	423
CONCLUSION	425
ODATA IN ASP.NET WEB APIS	427
ODATA	428

QUERY OPTIONS	429
SETUP SOLUTION	430
ODATA ENDPOINTS	432
\$TOP	438
\$FILTER	439
\$ORDERBY	443
\$ORDERBY WITH \$TOP	452
\$SKIP	454
STANDARD FILTER OPERATORS	456
STANDARD QUERY FUNCTIONS	458
PAGING	458
QUERY OPTIONS CONSTRAINTS	460
ALLOWEDQUERYOPTIONS	460
ALLOWEDORDERBYPROPERTIES	462
ALLOWEDLOGICALOPERATORS	465
ALLOWEDARITHMETICOPERATORS	465
CONCLUSION	466
CREATING SELF HOSTED ASP.NET WEB API WITH CRUD OPERATIONS IN VISUAL STUDIO 2010	468
WEBAPI PROJECT	469
WEBAPI TEST CLIENT	486
CONCLUSION	502
INDEX	503

About Author

Akhil Mittal is a Microsoft MVP, C# Corner MVP, a Code project MVP, blogger, programmer by heart and currently working as a Sr. Analyst in **Magic Software** and have an experience of more than 9 years in C#.Net. He is a B.Tech in Computer Science and holds a diploma in Information Security and Application Development. His work experience includes Development of Enterprise Applications using C#, .Net and SQL Server, Analysis as well as Research and Development. He is a MCP in Web Applications (MCTS-70-528, MCTS-70-515) and .Net Framework 2.0 (MCTS-70-536). Visit his personal blog **CodeTeddy** for more informative articles.



Akhil Mittal

Software)

Sr. Analyst (Magic

Enterprise-level application architecture with Web APIs using Entity Framework, Generic Repository pattern and Unit of Work

REST

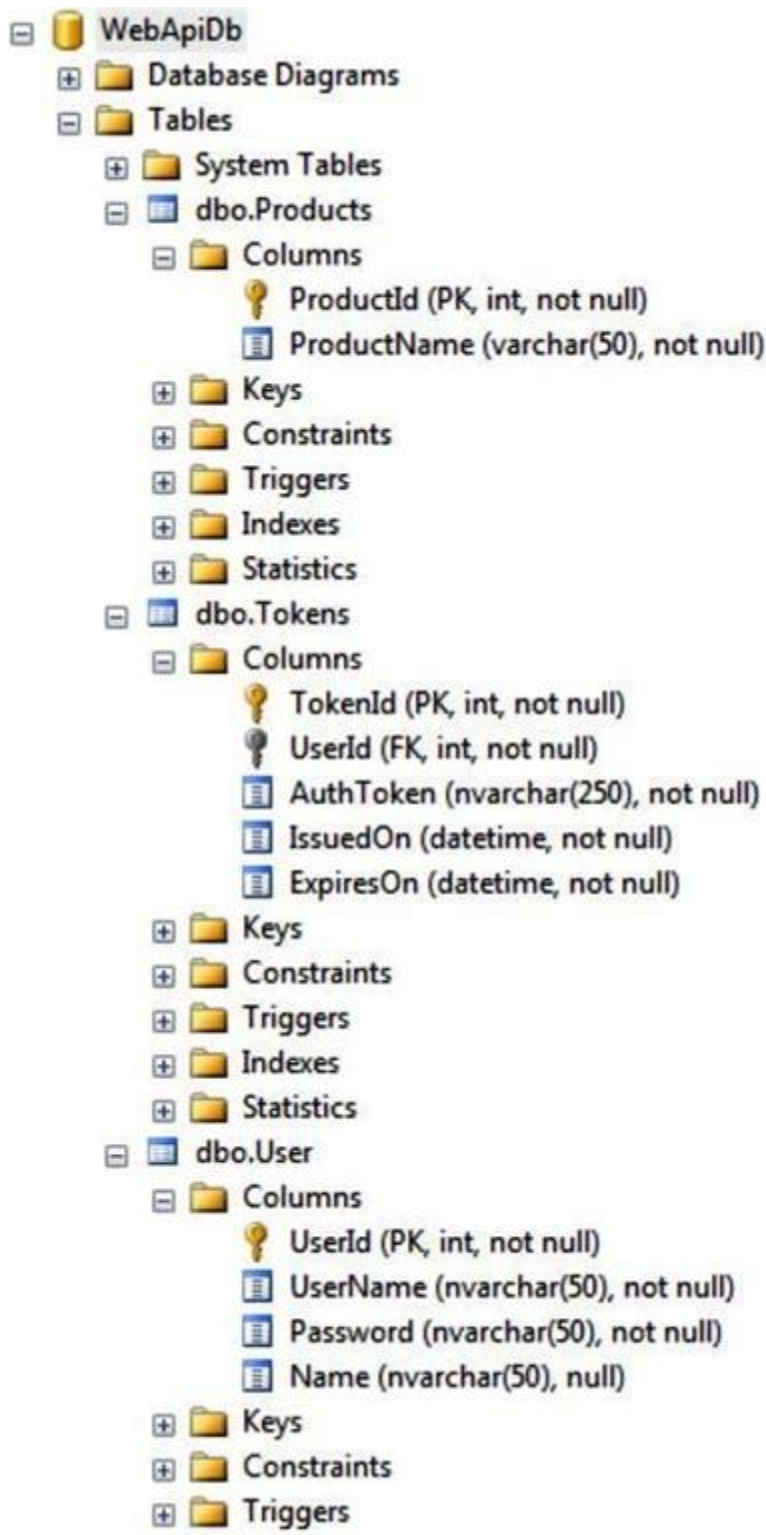
Here is an excerpt from the Wikipedia:

“Unlike SOAP-based web services, there is no "official" standard for **RESTful** web APIs. This is because REST is an architectural style, whereas SOAP is a protocol. Even though REST is not a standard per se, most **RESTful** implementations make use of standards such as HTTP, URI, JSON and XML.”

I agree to it. Let's do some coding.

Setup database

I am using SQL Server 2008 as a database server. I have provided the SQL scripts to create the database in SQL Server; you can use it to create one. I have given WebApiDb as database name. Database contains three tables for now; they are Products, Tokens and User. In this tutorial we'll only be dealing with a product table to do CRUD operations using Web API and Entity Framework. We'll use Tokens and Users in future chapter. For those who fail to create database using scripts, here is the structure you can use:

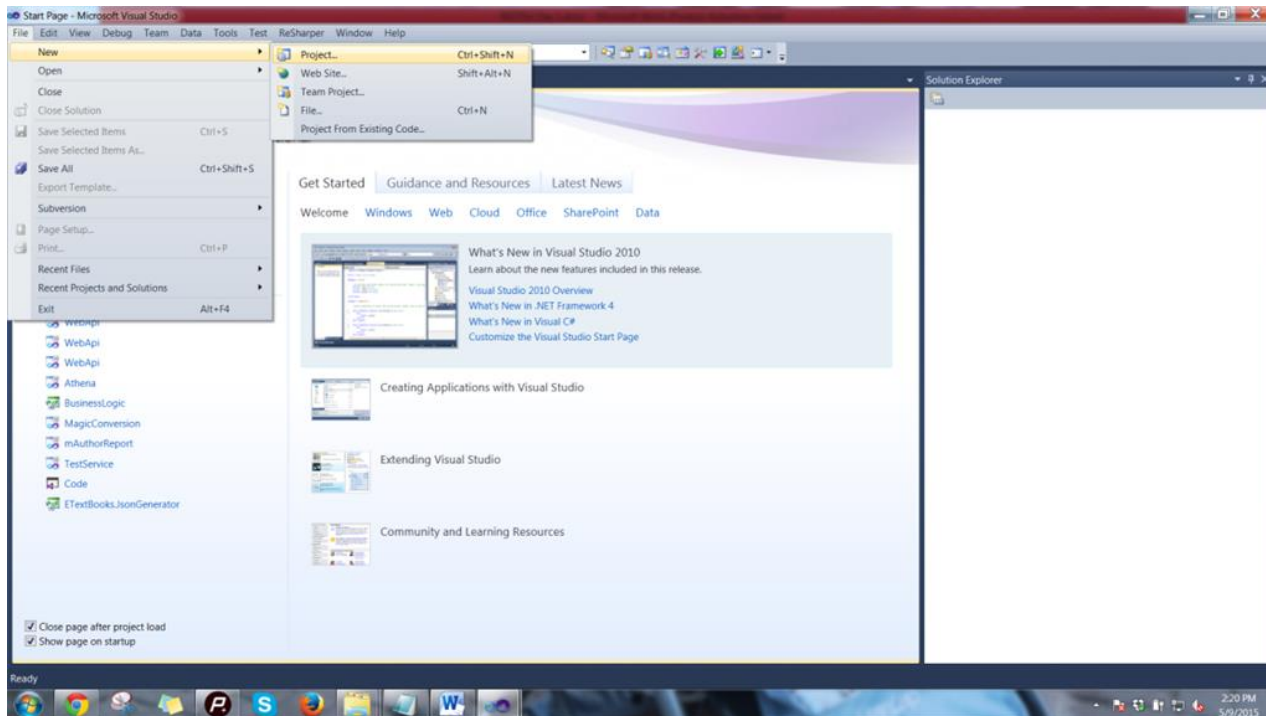


Web API project

Open your Visual Studio. I am using Visual Studio 2010, you can use Visual Studio version 2010 or above.

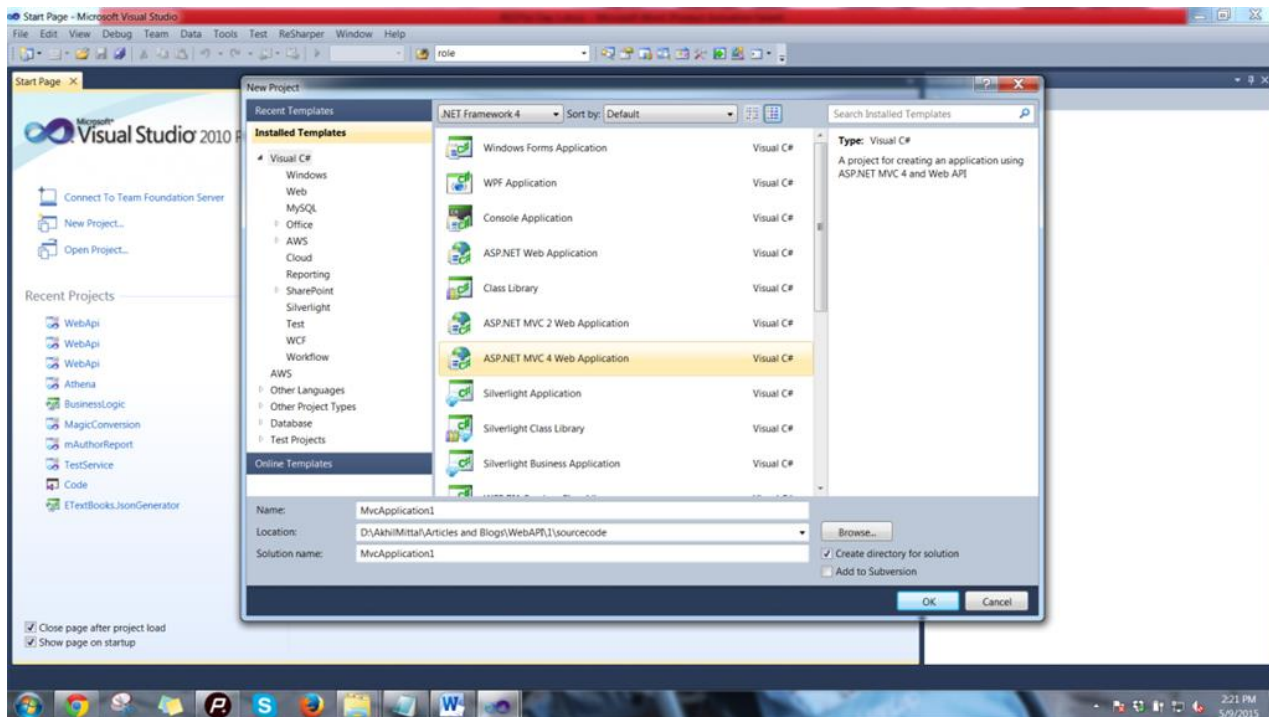
Step 1

Create a new Project in your Visual Studio as in the following:



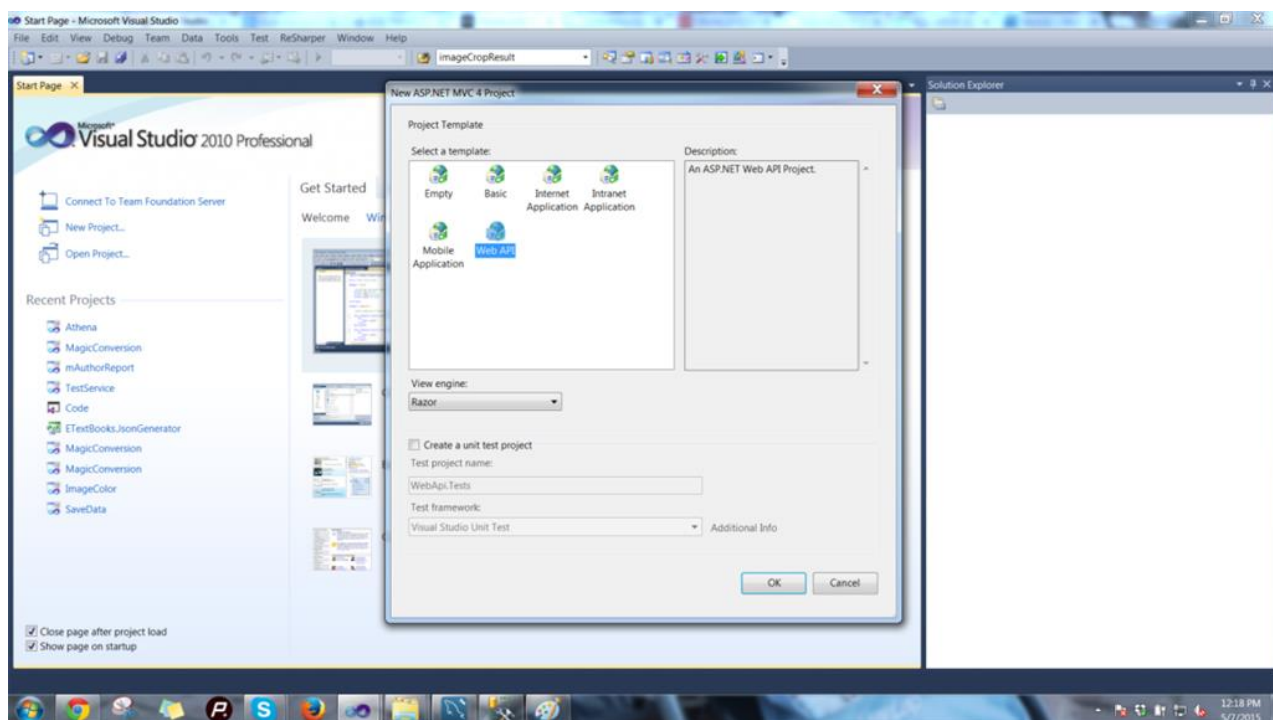
Step 2

There after, choose to create an ASP.Net MVC 4 Web Application and provide it the name of your choice, I used **WebAPI**.



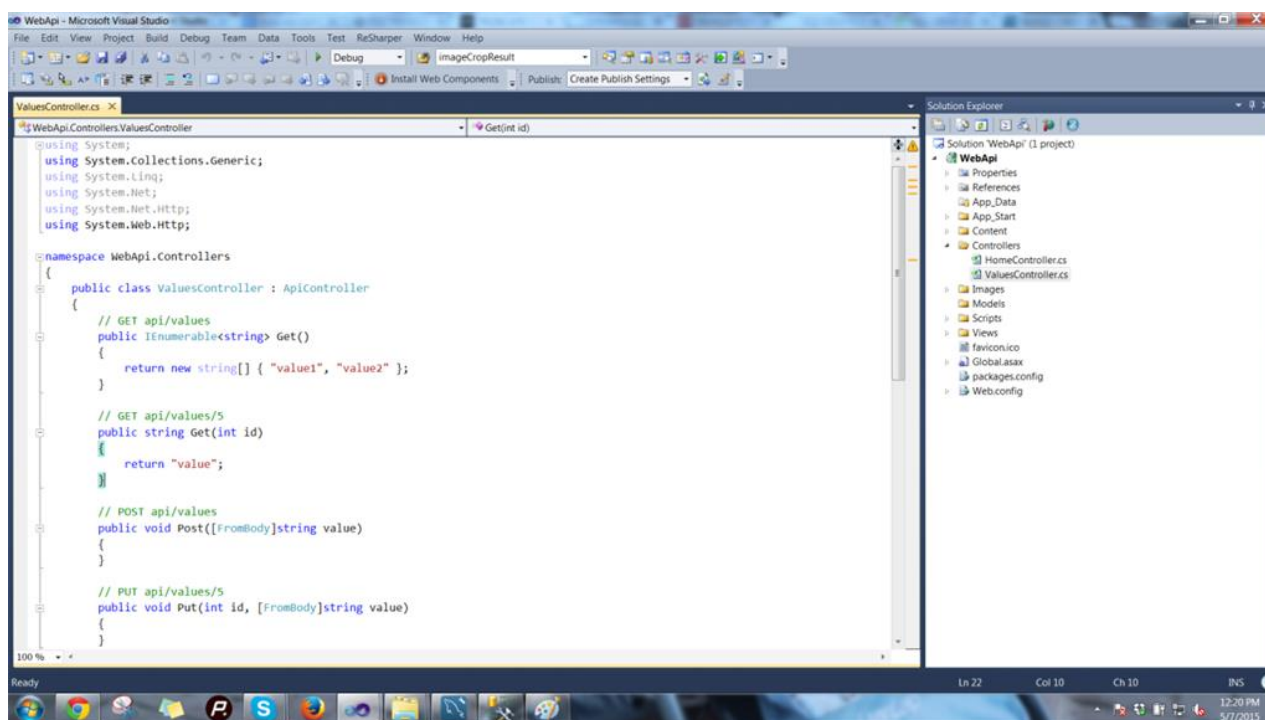
Step 3

Out of various type of project templates shown to you, choose Web API project as in the following:



Once done, you'll get a project structure like the one shown below, with a default Home and Values controller.

You can choose to delete this ValuesController, since we'll be using our own controller to learn.



Setup Data Access Layer

Let's setup our data access layer first. We'll be using Entity Framework 5.0 to talk to the database. We'll use the Generic Repository Pattern and Unit of Work pattern to standardize our layer.

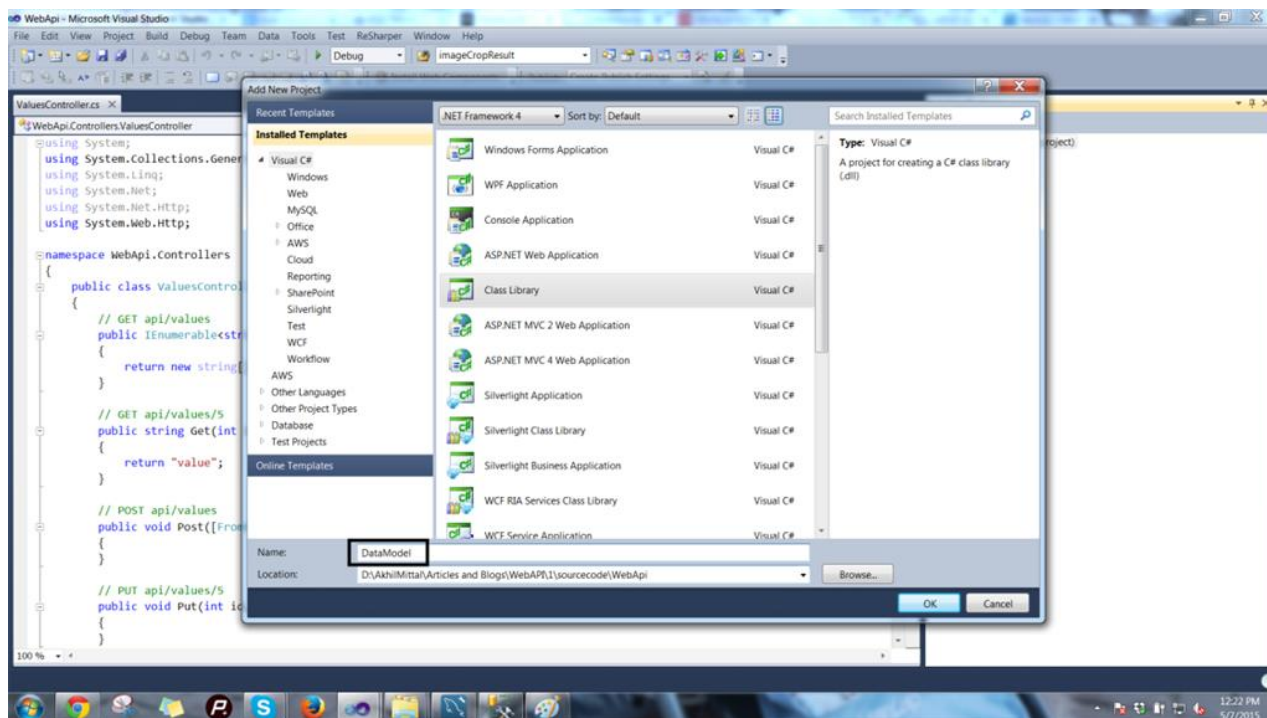
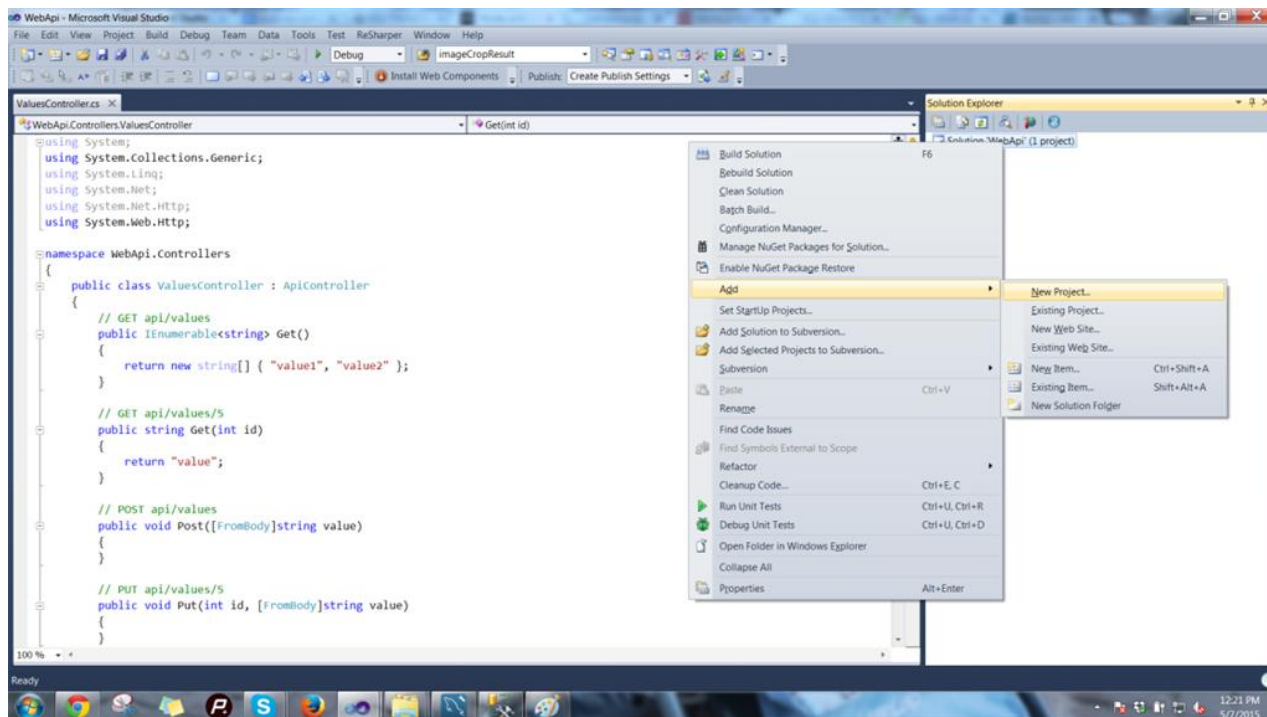
Let's have a look at the standard definition of Entity Framework given by Microsoft:

“The Microsoft ADO.NET Entity Framework is an Object/Relational Mapping (ORM) framework that enables developers to work with relational data as domain-specific objects, eliminating the need for most of the data access plumbing code that developers usually need to write. Using the Entity Framework, developers issue queries using LINQ, then retrieve and manipulate data as strongly typed objects. The Entity Framework's ORM implementation provides services like change tracking, identity resolution, lazy loading and query translation so that developers can focus on their application-specific business logic rather than the data access fundamentals.”

In simple language, Entity Framework is an Object/Relational Mapping (ORM) framework. It is an enhancement to ADO.NET, an upper layer to ADO.NET that gives developers an automated mechanism for accessing and storing the data in the database.

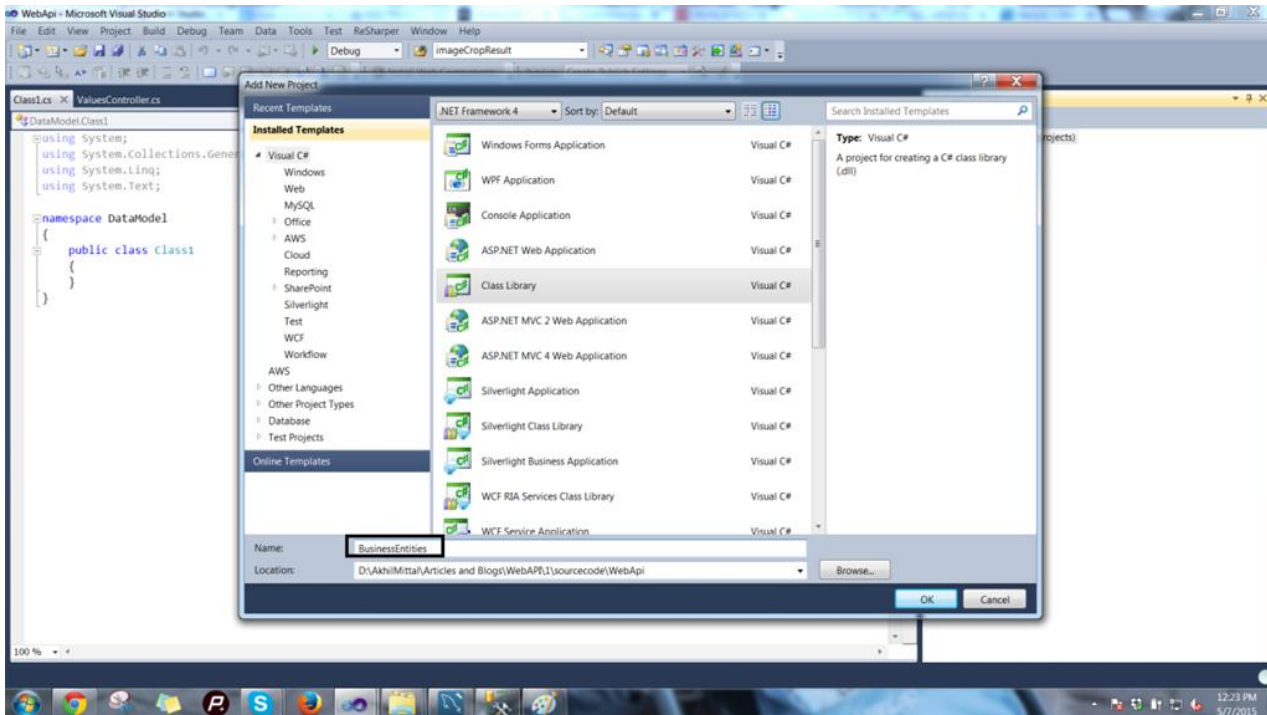
Step 1

Create a new class library in your Visual Studio and name it DataModel as shown below:



Step 2

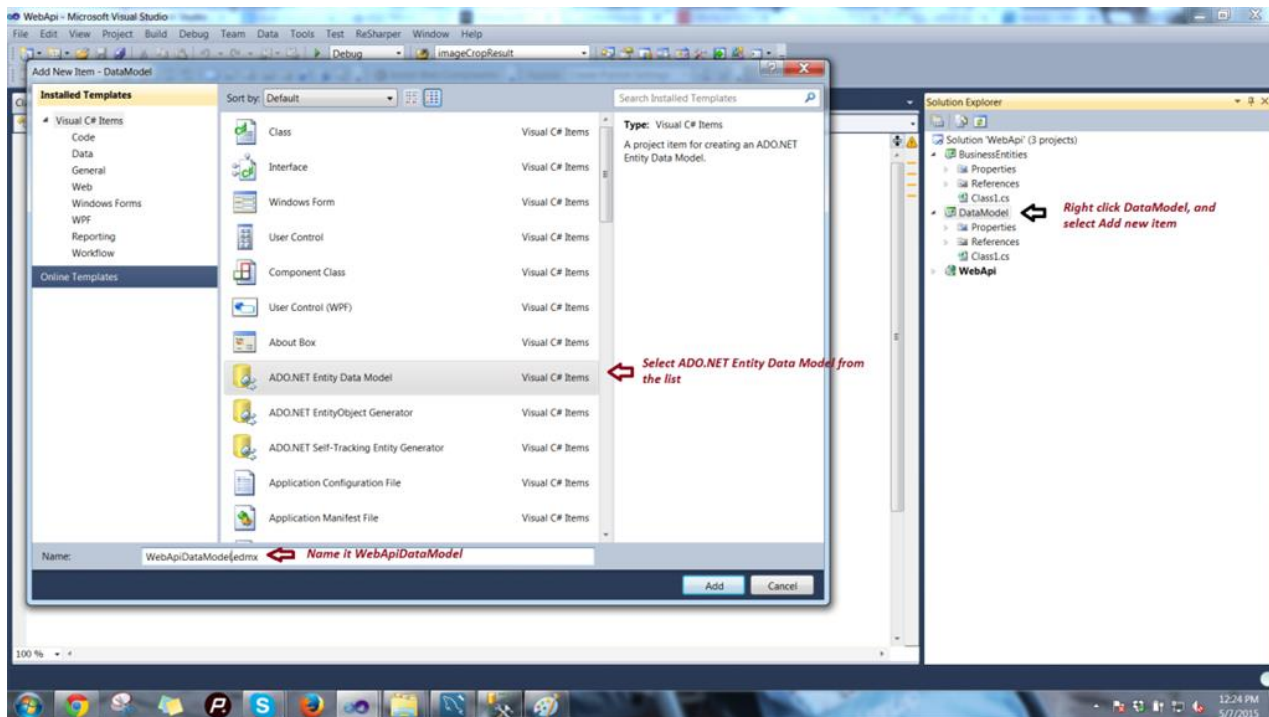
In the same way, create one more project, again a class library, and call it BusinessEntities as in the following:



I'll explain the use of this class library soon.

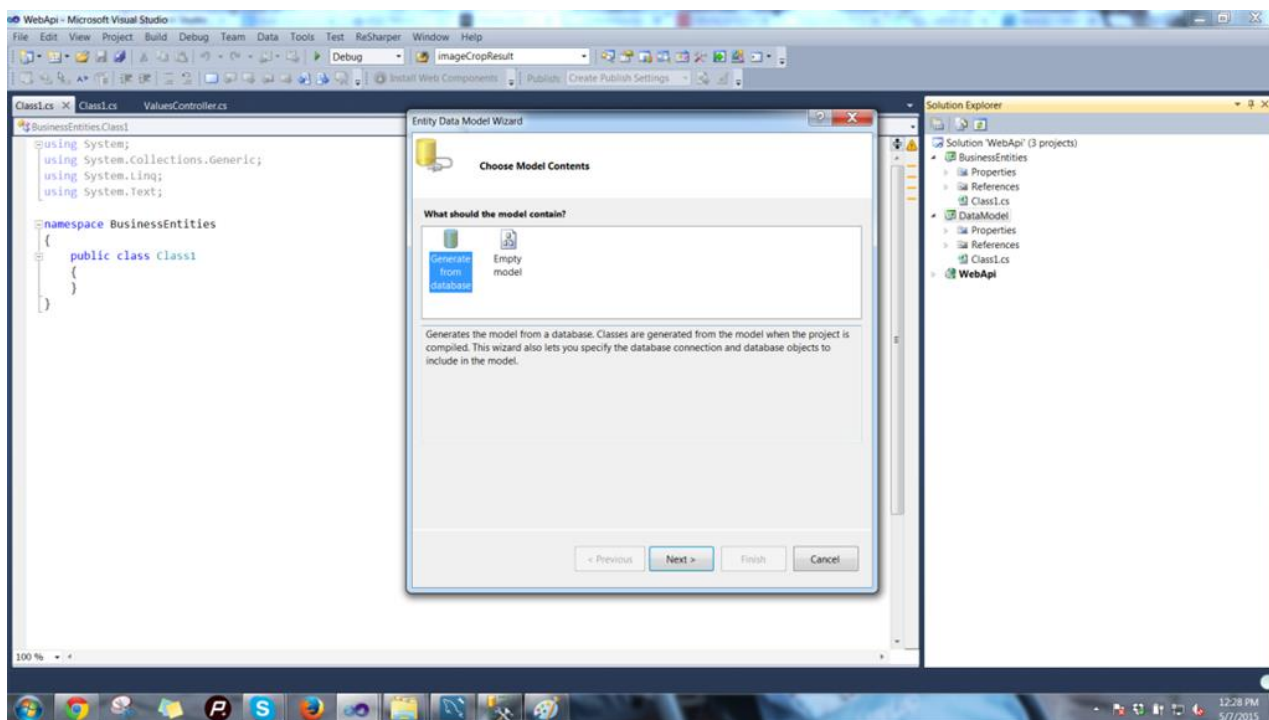
Step 3

Move on to your DataModel project. Right-click on it and add a new item in the list shown, choose ADO.Net Data Model and name it WebApiDataModel.edmx.

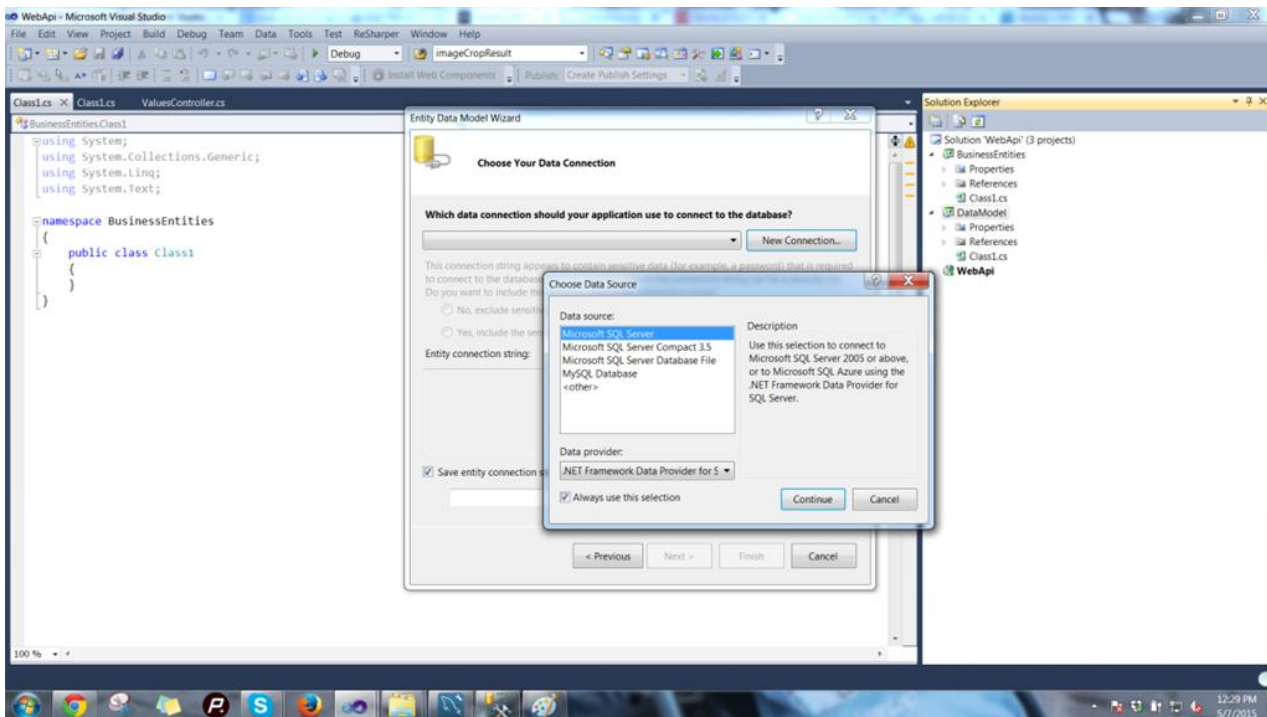


The file .edmx will contain the database information of our database that we created earlier, let's set this up.

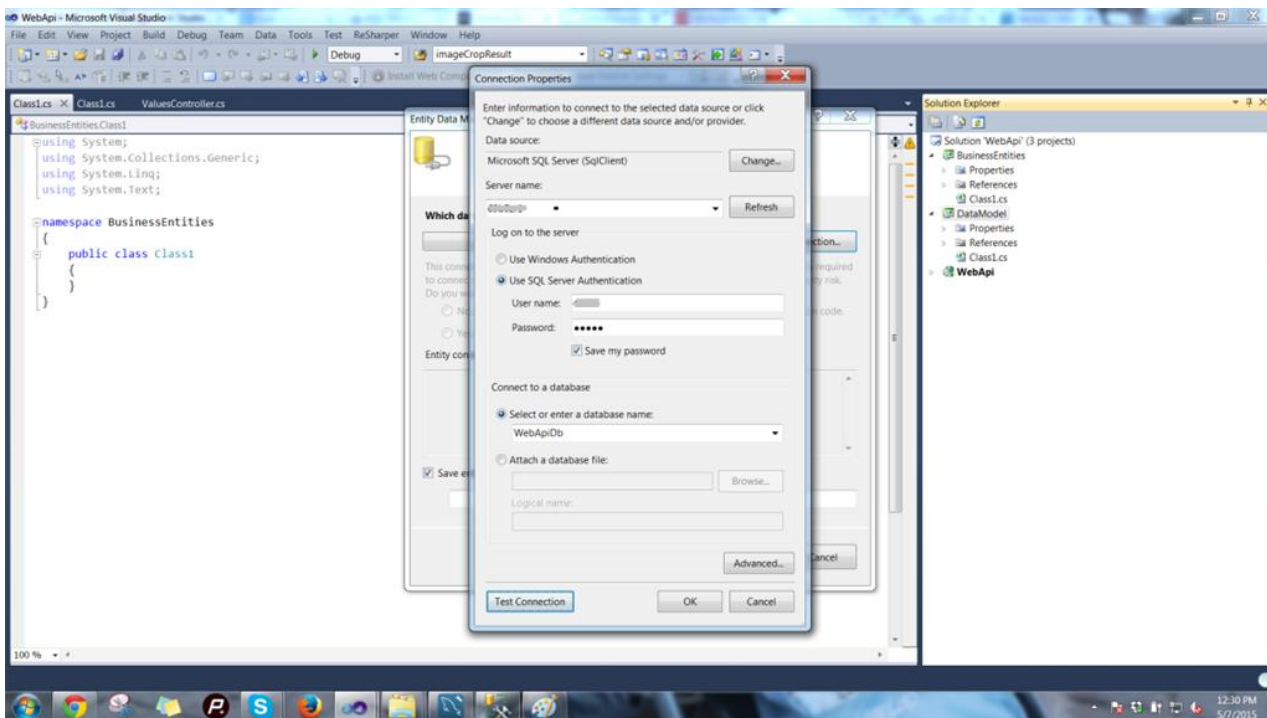
You'll be presented a wizard as follows:



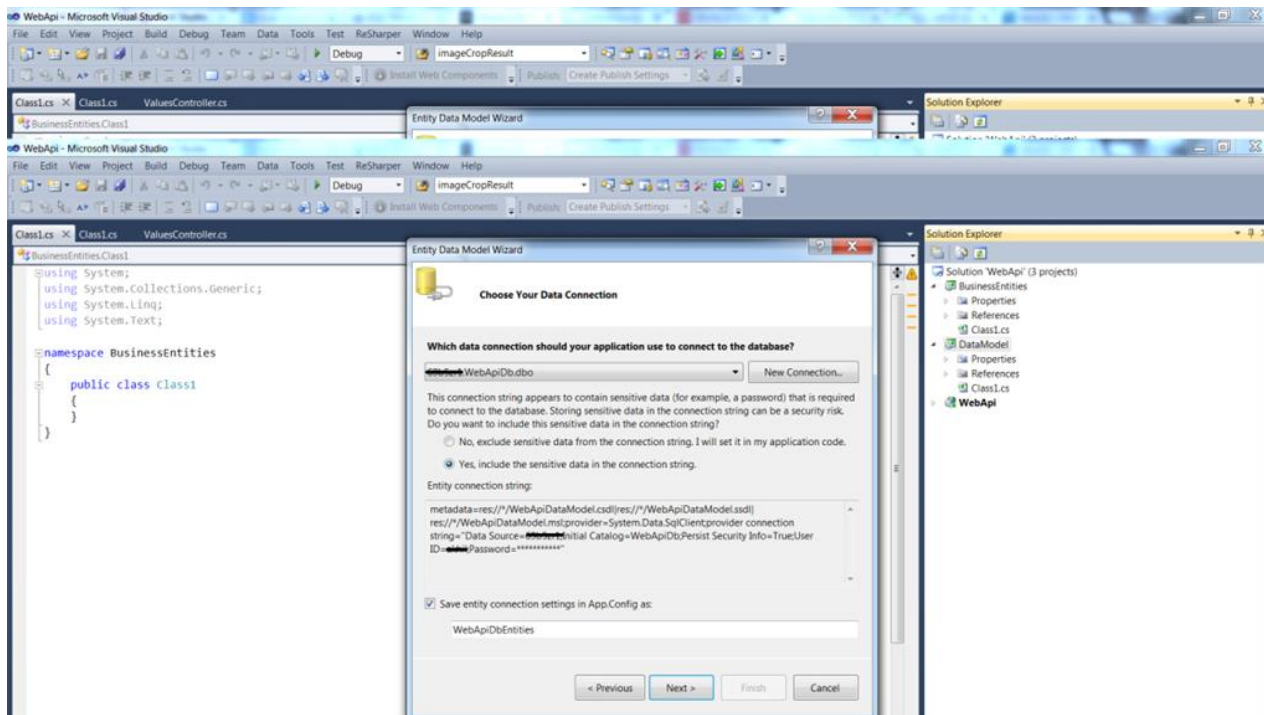
Choose generate from database. Choose Microsoft SQL Server as shown in the following image:



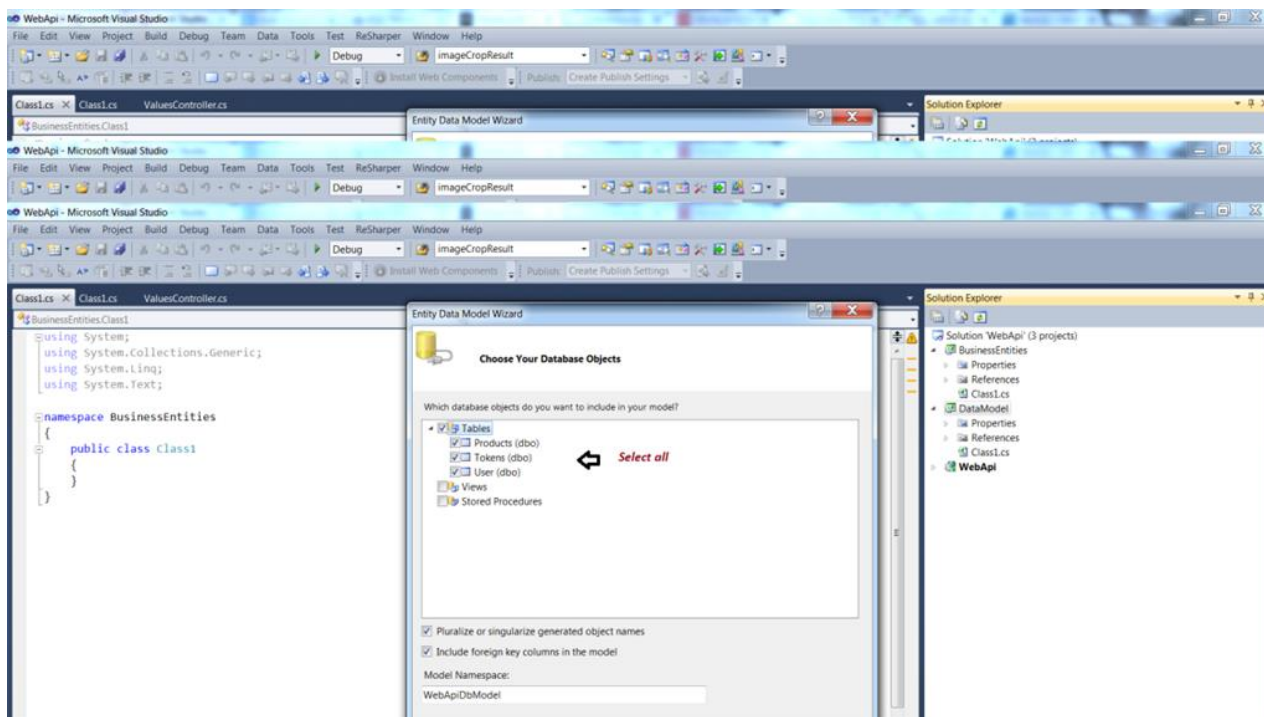
Click Continue, then provide the credentials for your database, in other words WebAPIdb and connect to it as in the following:



You'll get a screen showing the connection string of the database we chose as in the following:

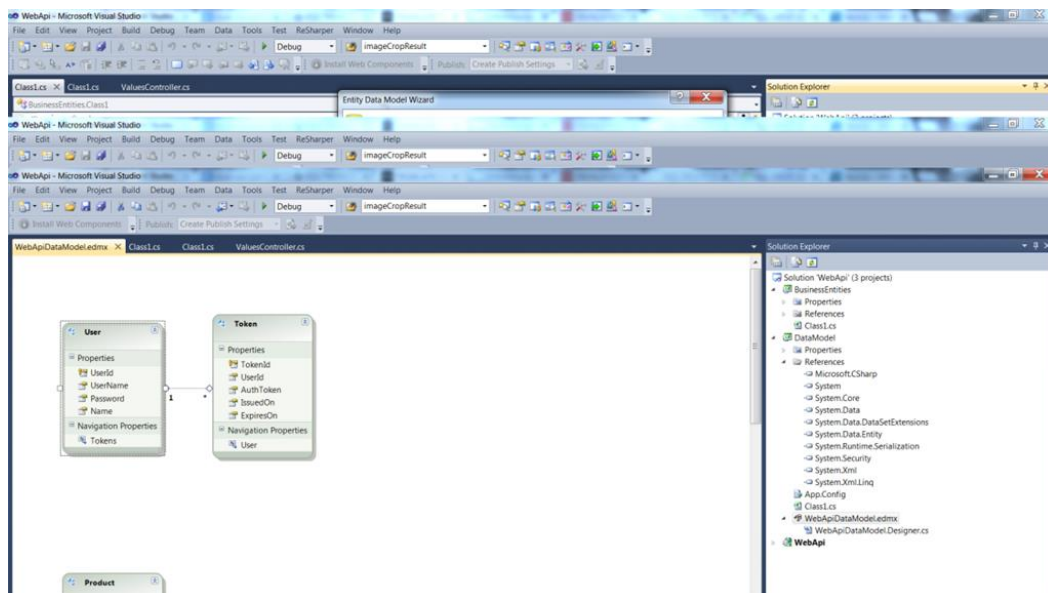


Provide the name of the connection string as WebApiDbEntities and click Next.



Choose all the database objects, check all the check boxes and provide a name for the model. I gave it the name WebApiDbModel.

Once you finish this wizard, you'll get the schema ready in your datamodel project as follows:

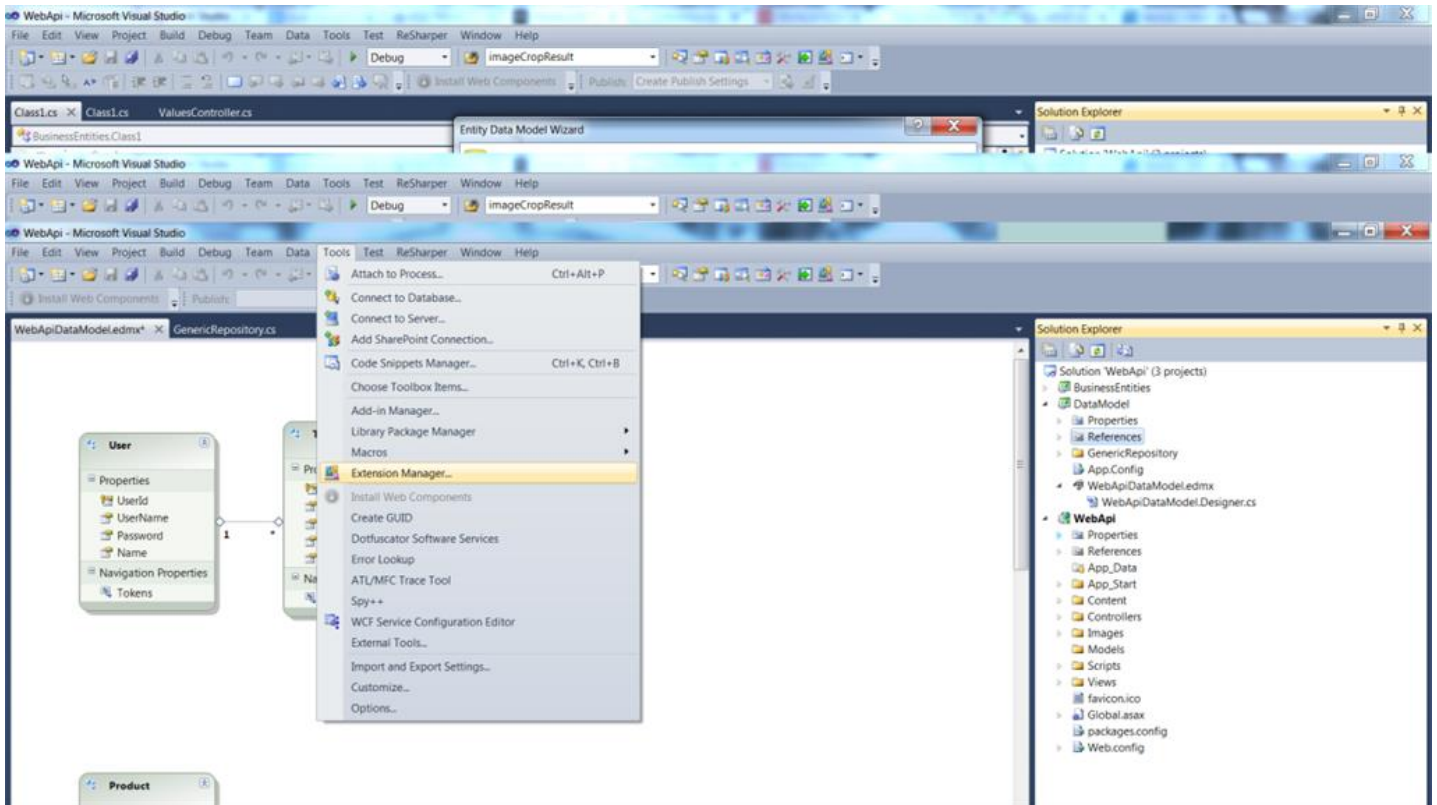


We've got our schema in-place using Entity Framework. But a bit of work remains. We need our data context class and entities through which we'll communicate with the database.

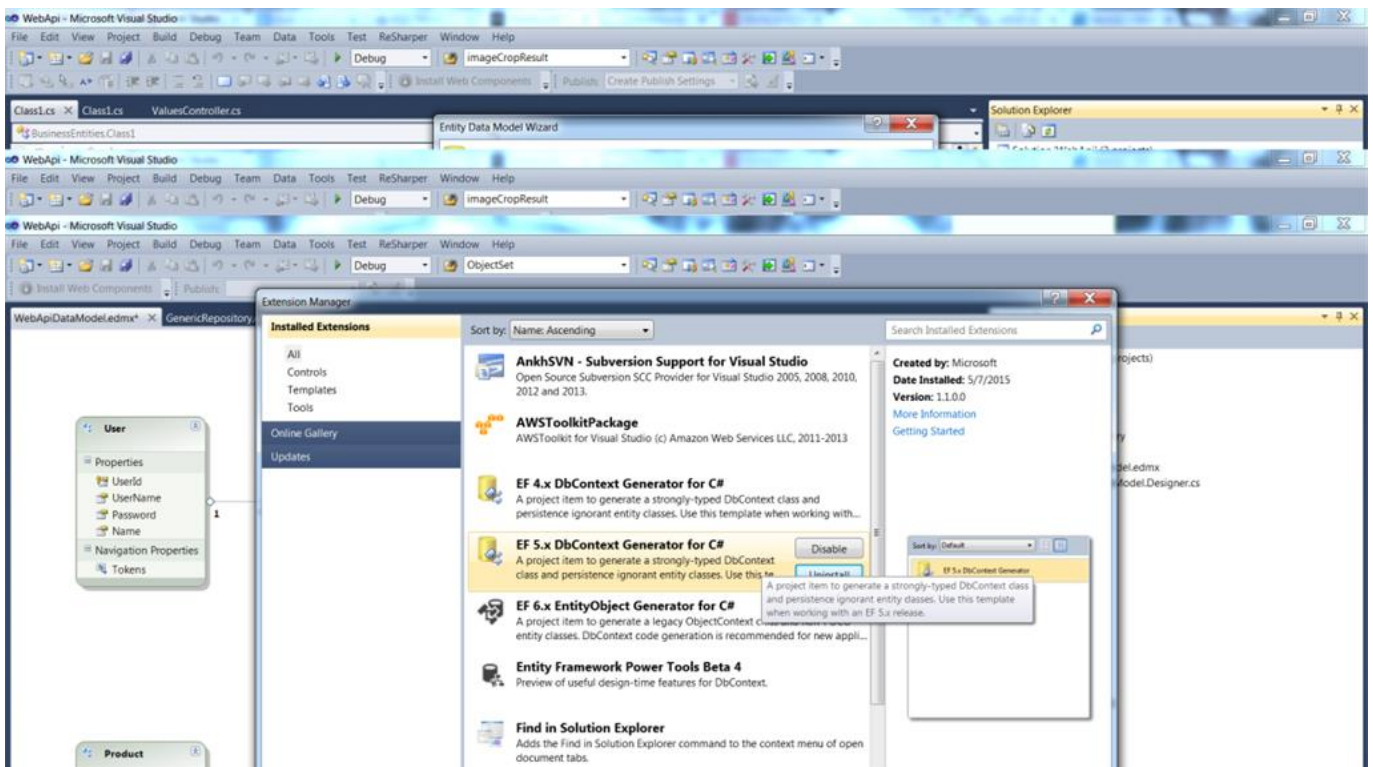
So, moving on to the next step.

Step 4

Click on Tools in Visual Studio and open the Extension Manager. We need to get a db context generator for our datamodel. We can also do it using the default code generation item by right-clicking in the edmx view and add a code generation item, but that will generate an object context class that is heavier than the db context. I want a lightweight db context class to be created, so we'll use the Extension Manager to add a package and then create a db context class.



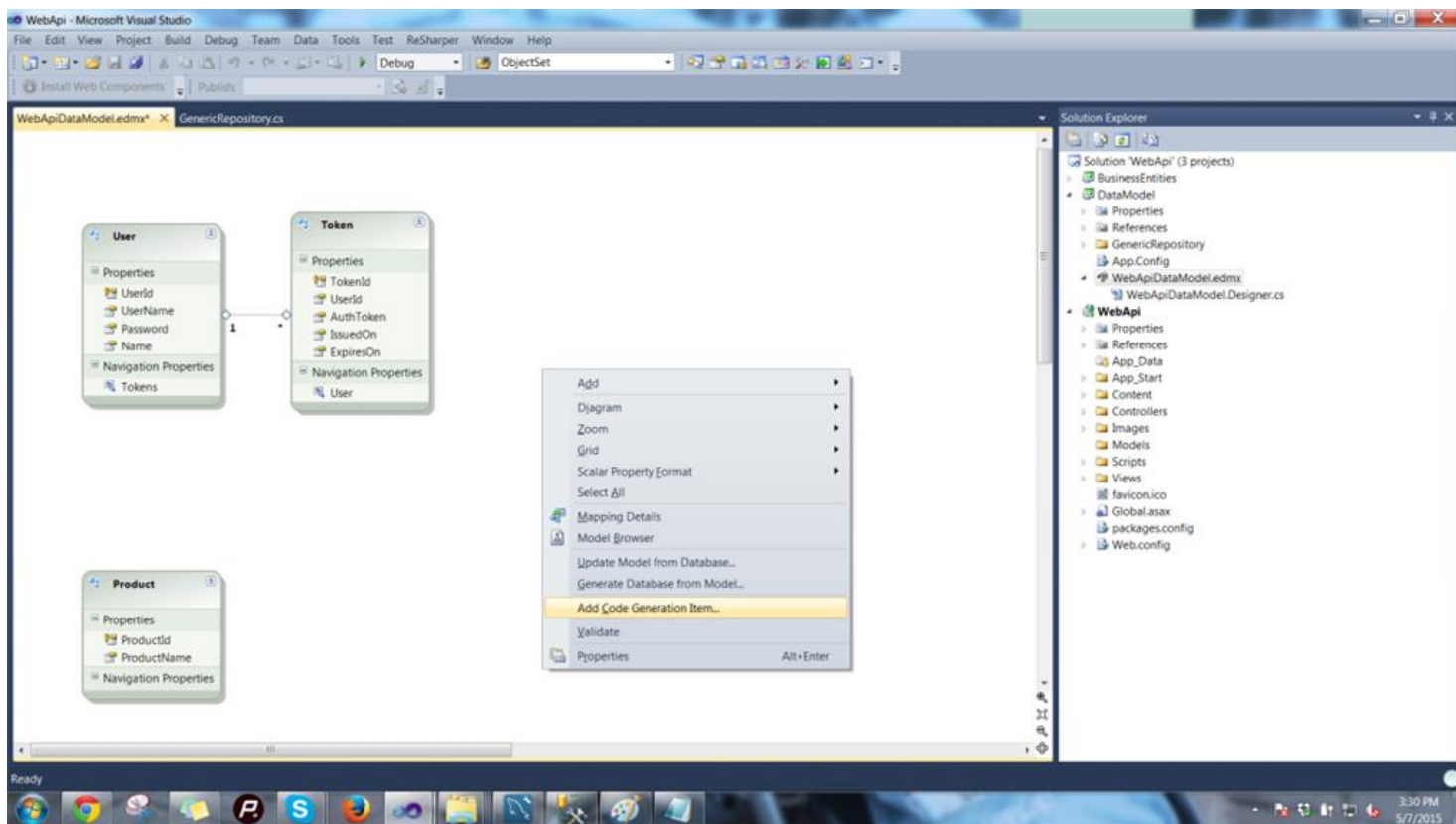
Search for Entity Framework Db context generator in the online gallery and select the one for EF 5.x as in the following:



I guess you need to restart Visual Studio to get that into your templates.

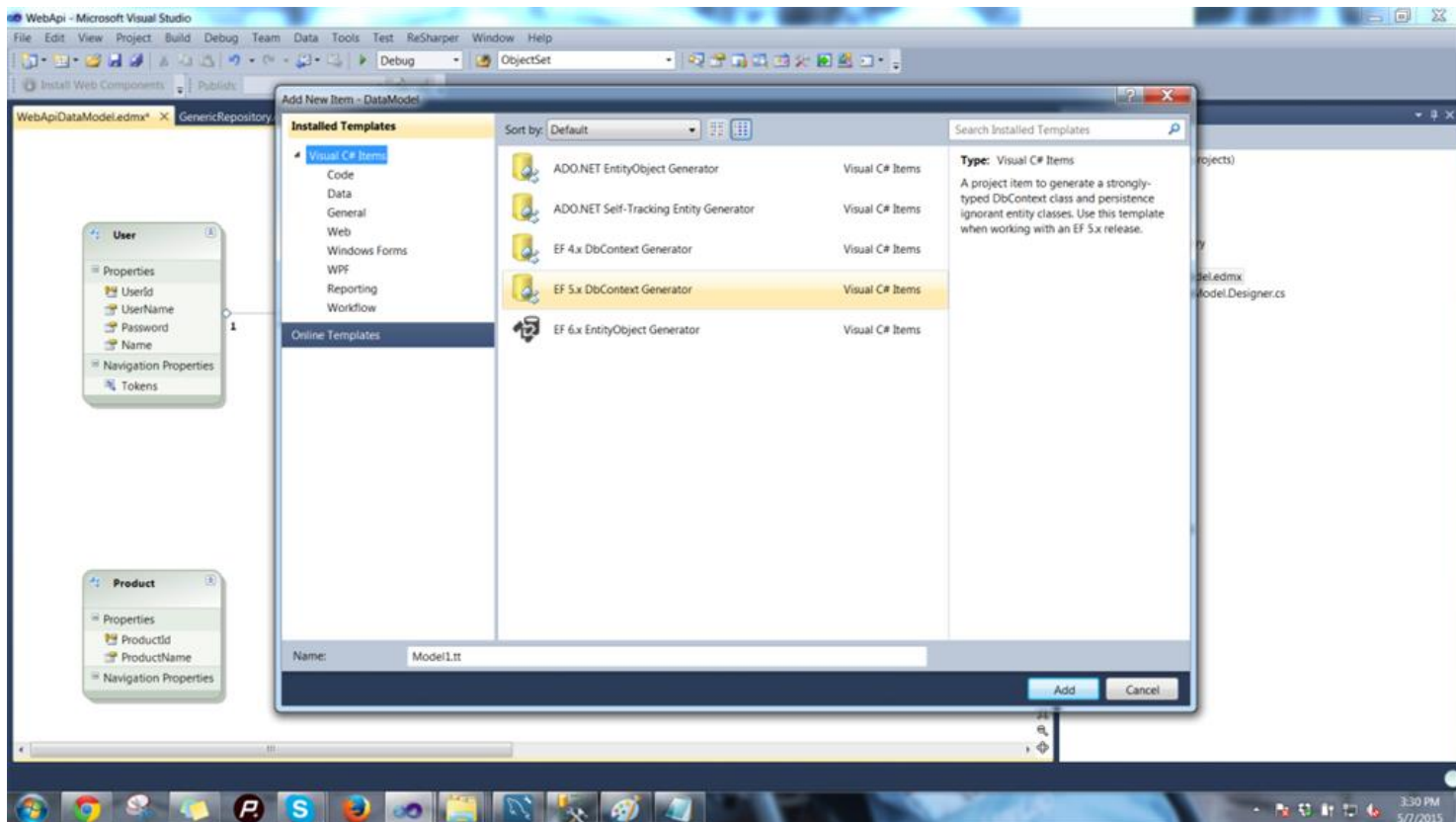
Step 5

Now right-click in the .edmx file schema designer and choose “Add Code Generation Item...”

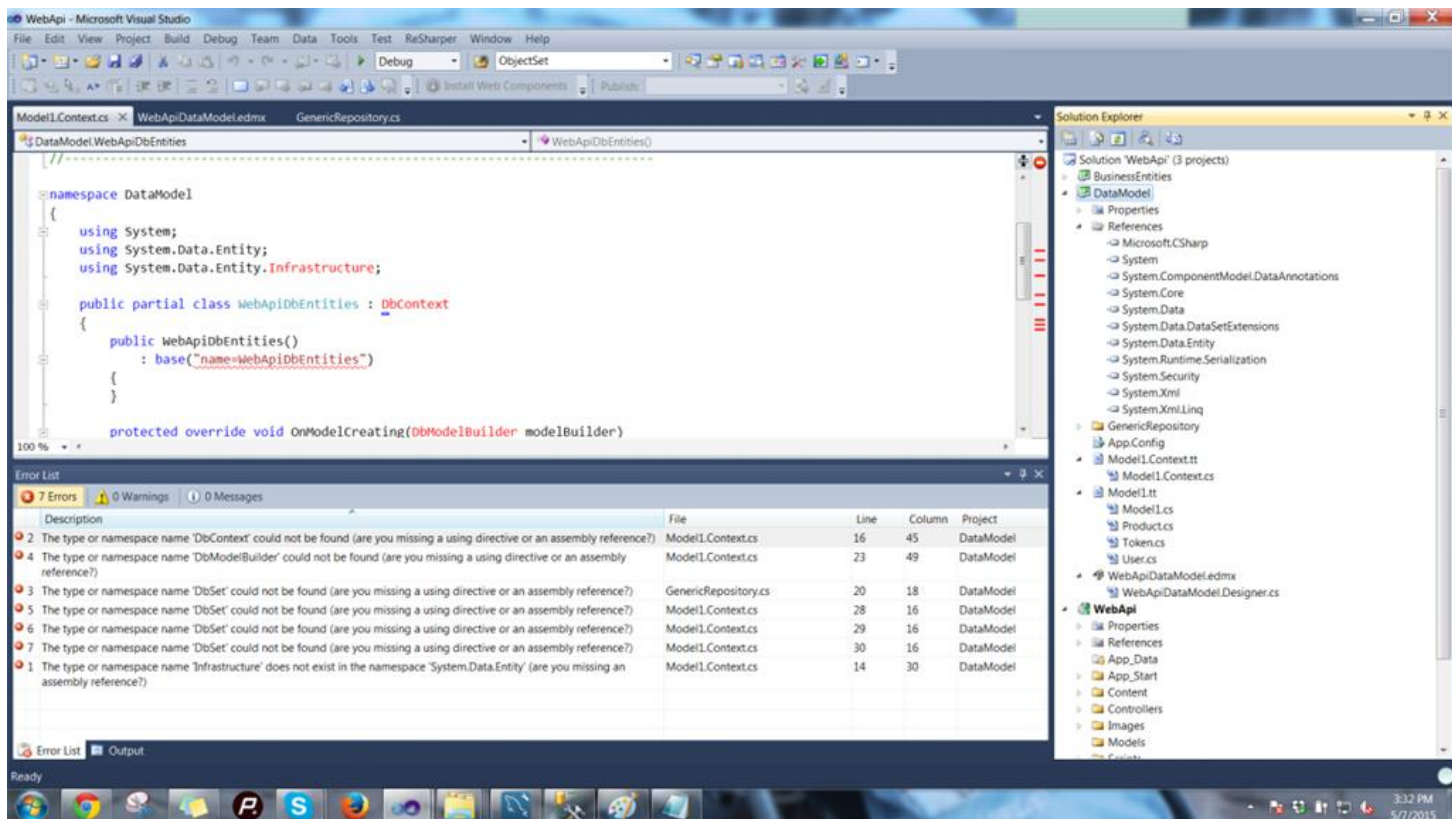


Step 6

Now you'll see that we have the template for the extension that we added, select that EF 5.x DbContext Generator and click Add.



After adding this we'll get the db context class and its properties. This class is responsible for all database transactions that we need to do, so our structure looks as shown below.

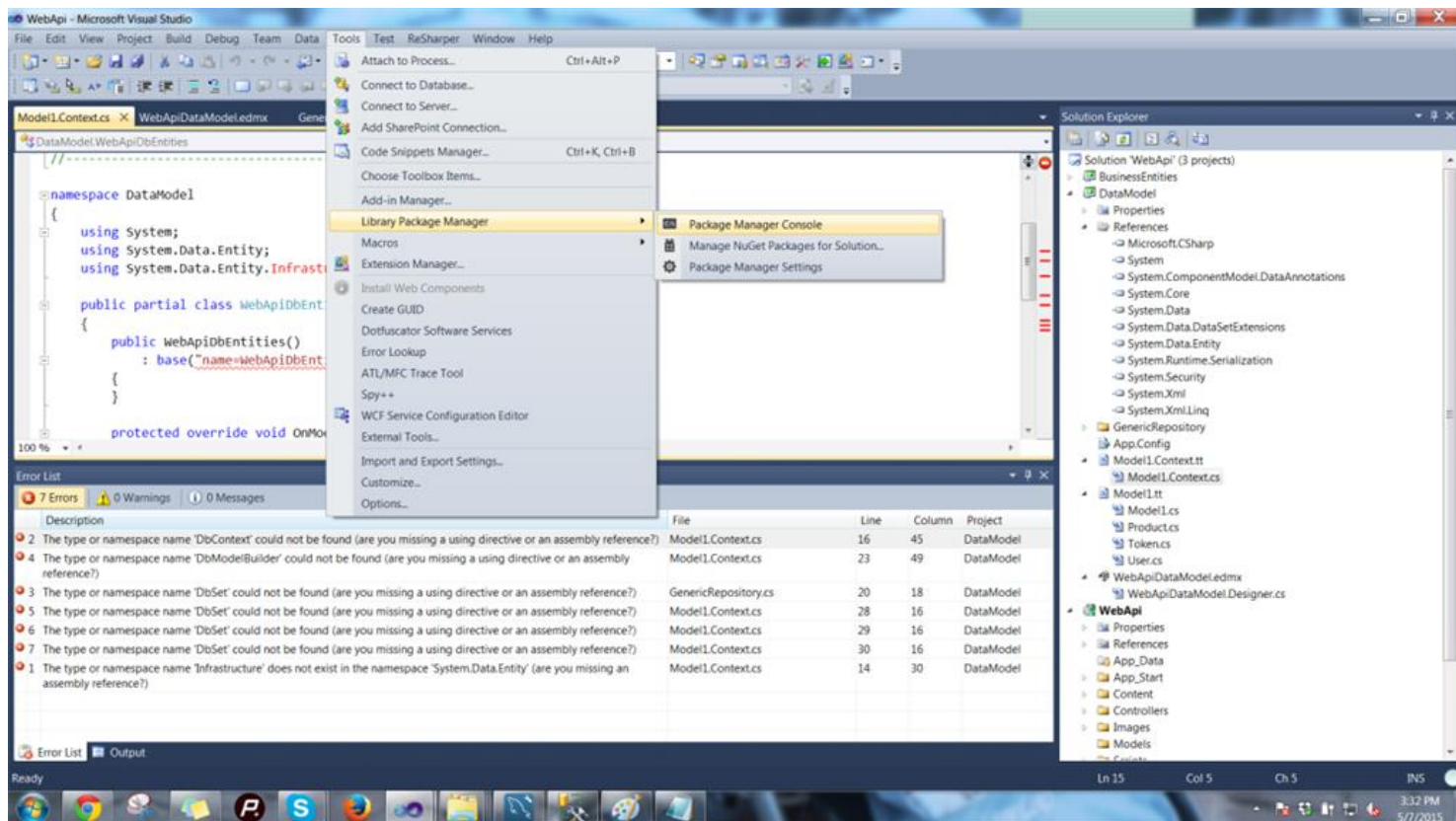


Wow, we ended up with errors. But we got our db context class and our entity models. You can see them in our DataModel project. Errors? Nothing to worry about, it's just we did not reference Entity Framework in our project. We'll do it the right away.

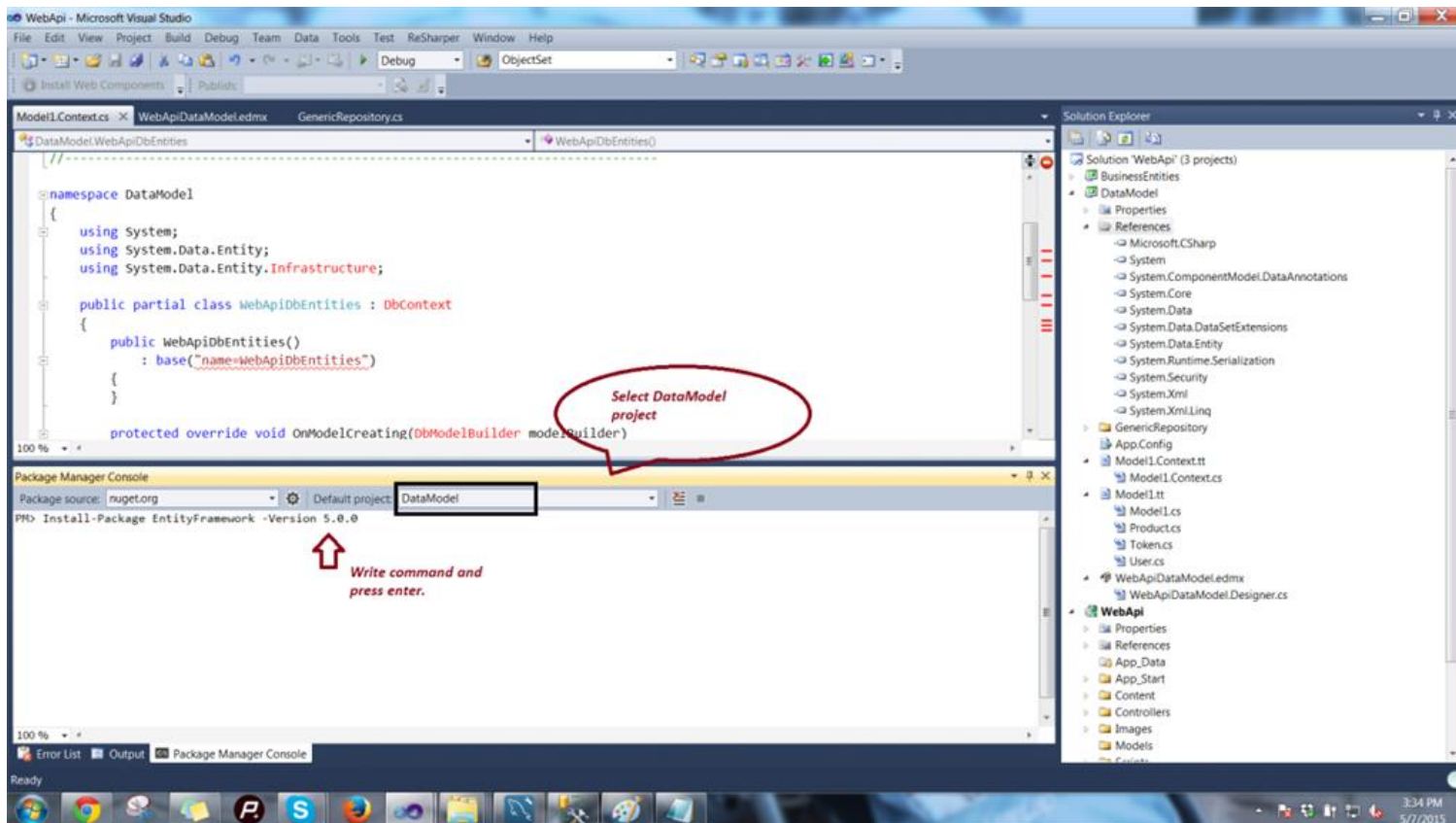
Step 7

Go to Tools -> Library Packet Manager -> Packet Manager Console.

You'll get the console in the bottom-left of Visual Studio.



Select dataModel project and provide the command **“Install-Package EntityFramework – Version 5.0.0”** to install Entity Framework 5 in our DataModel project.



Press Enter. And all the errors become resolved.

Generic Repository and Unit of Work

Just to list the benefits of the Repository Pattern:

- It centralizes the data logic or Web service access logic.
- It provides a substitution point for the unit tests.
- It provides a flexible architecture that can be adapted as the overall design of the application evolves.

We'll create a generic repository that works for all our entities. Creating repositories for each and every entity may result in lots of duplicate code in large projects.

Step 1

Add a folder named `GenericRepository` to the `DataModel` project and to that folder add a class named `Generic Repository`. Add the following code to that class that serves as a template-based generic code for all the entities that will interact with the database as in the following:

1. `#region Using Namespaces...`


```

2.
3. using System;
4. using System.Collections.Generic;
5. using System.Data;
6. using System.Data.Entity;
7. using System.Linq;
8.
9. #endregion
10.
11. namespace DataModel.GenericRepository
12. {
13.     /// <summary>
14.     /// Generic Repository class for Entity Operations
15.     /// </summary>
16.     /// <typeparam name="TEntity"></typeparam>
17.     public class GenericRepository<TEntity> where TEntity : class
18.     {
19.         #region Private member variables...
20.         internal WebApiDbEntities Context;
21.         internal DbSet<TEntity> DbSet;
22.         #endregion
23.
24.         #region Public Constructor...
25.         /// <summary>
26.         /// Public Constructor, initializes privately declared local variables.
27.         /// </summary>
28.         /// <param name="context"></param>
29.         public GenericRepository(WebApiDbEntities context)
30.         {
31.             this.Context = context;
32.             this.DbSet = context.Set<TEntity>();
33.         }
34.         #endregion
35.
36.         #region Public member methods...
37.

```



```
38.    /// <summary>
39.    /// generic Get method for Entities
40.    /// </summary>
41.    /// <returns></returns>
42.    public virtual IEnumerable<TEntity> Get()
43.    {
44.        IQueryable<TEntity> query = DbSet;
45.        return query.ToList();
46.    }
47.
48.    /// <summary>
49.    /// Generic get method on the basis of id for Entities.
50.    /// </summary>
51.    /// <param name="id"></param>
52.    /// <returns></returns>
53.    public virtual TEntity GetByID(object id)
54.    {
55.        return DbSet.Find(id);
56.    }
57.
58.    /// <summary>
59.    /// generic Insert method for the entities
60.    /// </summary>
61.    /// <param name="entity"></param>
62.    public virtual void Insert(TEntity entity)
63.    {
64.        DbSet.Add(entity);
65.    }
66.
67.    /// <summary>
68.    /// Generic Delete method for the entities
69.    /// </summary>
70.    /// <param name="id"></param>
71.    public virtual void Delete(object id)
72.    {
73.        TEntity entityToDelete = DbSet.Find(id);
```



```
74.     Delete(entityToDelete);
75. }
76.
77.     /// <summary>
78.     /// Generic Delete method for the entities
79.     /// </summary>
80.     /// <param name="entityToDelete"></param>
81.     public virtual void Delete(TEntity entityToDelete)
82.     {
83.         if (Context.Entry(entityToDelete).State == EntityState.Detached)
84.         {
85.             DbSet.Attach(entityToDelete);
86.         }
87.         DbSet.Remove(entityToDelete);
88.     }
89.
90.     /// <summary>
91.     /// Generic update method for the entities
92.     /// </summary>
93.     /// <param name="entityToUpdate"></param>
94.     public virtual void Update(TEntity entityToUpdate)
95.     {
96.         DbSet.Attach(entityToUpdate);
97.         Context.Entry(entityToUpdate).State = EntityState.Modified;
98.     }
99.
100.        /// <summary>
101.        /// generic method to get many record on the basis of a condition.
102.        /// </summary>
103.        /// <param name="where"></param>
104.        /// <returns></returns>
105.        public virtual IEnumerable<TEntity> GetMany(Func<TEntity, bool> where)
106.        {
107.            return DbSet.Where(where).ToList();
108.        }
```



```

109.
110.    /// <summary>
111.    /// generic method to get many record on the basis of a condition but que
ry able.
112.    /// </summary>
113.    /// <param name="where"></param>
114.    /// <returns></returns>
115.    public virtual IQueryable<TEntity> GetManyQueryable(Func<TEntity, bool
> where)
116.    {
117.        return DbSet.Where(where).AsQueryable();
118.    }
119.
120.    /// <summary>
121.    /// generic get method , fetches data for the entities on the basis of condit
ion.
122.    /// </summary>
123.    /// <param name="where"></param>
124.    /// <returns></returns>
125.    public TEntity Get(Func<TEntity, Boolean> where)
126.    {
127.        return DbSet.Where(where).FirstOrDefault<TEntity>();
128.    }
129.
130.    /// <summary>
131.    /// generic delete method , deletes data for the entities on the basis of co
ndition.
132.    /// </summary>
133.    /// <param name="where"></param>
134.    /// <returns></returns>
135.    public void Delete(Func<TEntity, Boolean> where)
136.    {
137.        IQueryable<TEntity> objects = DbSet.Where<TEntity>(where).AsQuerya
ble();
138.        foreach (TEntity obj in objects)
139.            DbSet.Remove(obj);

```



```

140.     }
141.
142.     /// <summary>
143.     /// generic method to fetch all the records from db
144.     /// </summary>
145.     /// <returns></returns>
146.     public virtual IEnumerable<TEntity> GetAll()
147.     {
148.         return DbSet.ToList();
149.     }
150.
151.     /// <summary>
152.     /// Include multiple
153.     /// </summary>
154.     /// <param name="predicate"></param>
155.     /// <param name="include"></param>
156.     /// <returns></returns>
157.     public IQueryable<TEntity> GetWithInclude(System.Linq.Expressions.Expr
        session<Func<TEntity, bool>> predicate, params string[] include)
158.     {
159.         IQueryable<TEntity> query = this.DbSet;
160.         query = include.Aggregate(query, (current, inc) => current.Include(inc));
161.
162.         return query.Where(predicate);
163.     }
164.
165.     /// <summary>
166.     /// Generic method to check if entity exists
167.     /// </summary>
168.     /// <param name="primaryKey"></param>
169.     /// <returns></returns>
170.     public bool Exists(object primaryKey)
171.     {
172.         return DbSet.Find(primaryKey) != null;
173.     }

```



```

174.      /// <summary>
175.      /// Gets a single record by the specified criteria (usually the unique identifier)
176.      /// </summary>
177.      /// <param name="predicate">Criteria to match on</param>
178.      /// <returns>A single record that matches the specified criteria</returns>
179.      public TEntity GetSingle(Func<TEntity, bool> predicate)
180.      {
181.          return DbSet.Single<TEntity>(predicate);
182.      }
183.
184.      /// <summary>
185.      /// The first record matching the specified criteria
186.      /// </summary>
187.      /// <param name="predicate">Criteria to match on</param>
188.      /// <returns>A single record containing the first record matching the specified criteria</returns>
189.      public TEntity GetFirst(Func<TEntity, bool> predicate)
190.      {
191.          return DbSet.First<TEntity>(predicate);
192.      }
193.
194.
195.      #endregion
196.  }
197.  }

```

Step 2

To give a heads up, the important responsibilities of Unit of Work are the following:

- To manage transactions.
- To order the database inserts deletes and updates.
- To prevent duplicate updates. Inside a single usage of a Unit of Work object, various parts of the code may mark the same Invoice object as changed, but the Unit of Work class will only issue a single UPDATE command to the database.

The value of using a Unit of Work pattern is to free the rest of our code from these concerns so that you can otherwise concentrate on the business logic.

Create a folder named UnitOfWork and add a class to that folder named UnitOfWork.cs.

Add GenericRepository properties for all the three entities that we got. The class also implements an IDisposable interface and its method Dispose to free up connections and objects. The class will be as follows.

```

1. #region Using Namespaces...
2.
3. using System;
4. using System.Collections.Generic;
5. using System.Diagnostics;
6. using System.Data.Entity.Validation;
7. using DataModel.GenericRepository;
8.
9. #endregion
10.
11. namespace DataModel.UnitOfWork
12. {
13.     /// <summary>
14.     /// Unit of Work class responsible for DB transactions
15.     /// </summary>
16.     public class UnitOfWork : IDisposable
17.     {
18.         #region Private member variables...
19.
20.         private WebApiDbEntities _context = null;
21.         private GenericRepository<User> _userRepository;
22.         private GenericRepository<Product> _productRepository;
23.         private GenericRepository<Token> _tokenRepository;
24.         #endregion
25.
26.         public UnitOfWork()
27.         {
28.             _context = new WebApiDbEntities();

```



```
29.     }
30.
31.     #region Public Repository Creation properties...
32.
33.     /// <summary>
34.     /// Get/Set Property for product repository.
35.     /// </summary>
36.     public GenericRepository<Product> ProductRepository
37.     {
38.         get
39.         {
40.             if (this._productRepository == null)
41.                 this._productRepository = new GenericRepository<Product>(_context);
42.             return _productRepository;
43.         }
44.     }
45.
46.     /// <summary>
47.     /// Get/Set Property for user repository.
48.     /// </summary>
49.     public GenericRepository<User> UserRepository
50.     {
51.         get
52.         {
53.             if (this._userRepository == null)
54.                 this._userRepository = new GenericRepository<User>(_context);
55.             return _userRepository;
56.         }
57.     }
58.
59.     /// <summary>
60.     /// Get/Set Property for token repository.
61.     /// </summary>
62.     public GenericRepository<Token> TokenRepository
63.     {
64.         get
```



```

65.     {
66.         if (this._tokenRepository == null)
67.             this._tokenRepository = new GenericRepository<Token>(_context);
68.         return _tokenRepository;
69.     }
70. }
71. #endregion
72.
73. #region Public member methods...
74. /// <summary>
75. /// Save method.
76. /// </summary>
77. public void Save()
78. {
79.     try
80.     {
81.         _context.SaveChanges();
82.     }
83.     catch (DbEntityValidationException e)
84.     {
85.
86.         var outputLines = new List<string>();
87.         foreach (var eve in e.EntityValidationErrors)
88.         {
89.             outputLines.Add(string.Format("{0}: Entity of type \"{1}\" in state \"{2}\" h
as the following validation errors:", DateTime.Now, eve.Entry.Entity.GetType().Name,
eve.Entry.State));
90.             foreach (var ve in eve.ValidationErrors)
91.             {
92.                 outputLines.Add(string.Format("-
Property: \"{0}\", Error: \"{1}\"", ve.PropertyName, ve.ErrorMessage));
93.             }
94.         }
95.         System.IO.File.AppendAllLines(@"C:\errors.txt", outputLines);
96.
97.         throw e;

```



```
98.     }
99.
100.    }
101.
102.    #endregion
103.
104.    #region Implementing IDiosposable...
105.
106.    #region private dispose variable declaration...
107.    private bool disposed = false;
108.    #endregion
109.
110.    /// <summary>
111.    /// Protected Virtual Dispose method
112.    /// </summary>
113.    /// <param name="disposing"></param>
114.    protected virtual void Dispose(bool disposing)
115.    {
116.        if (!this.disposed)
117.        {
118.            if (disposing)
119.            {
120.                Debug.WriteLine("UnitOfWork is being disposed");
121.                _context.Dispose();
122.            }
123.        }
124.        this.disposed = true;
125.    }
126.
127.    /// <summary>
128.    /// Dispose method
129.    /// </summary>
130.    public void Dispose()
131.    {
132.        Dispose(true);
133.        GC.SuppressFinalize(this);
```

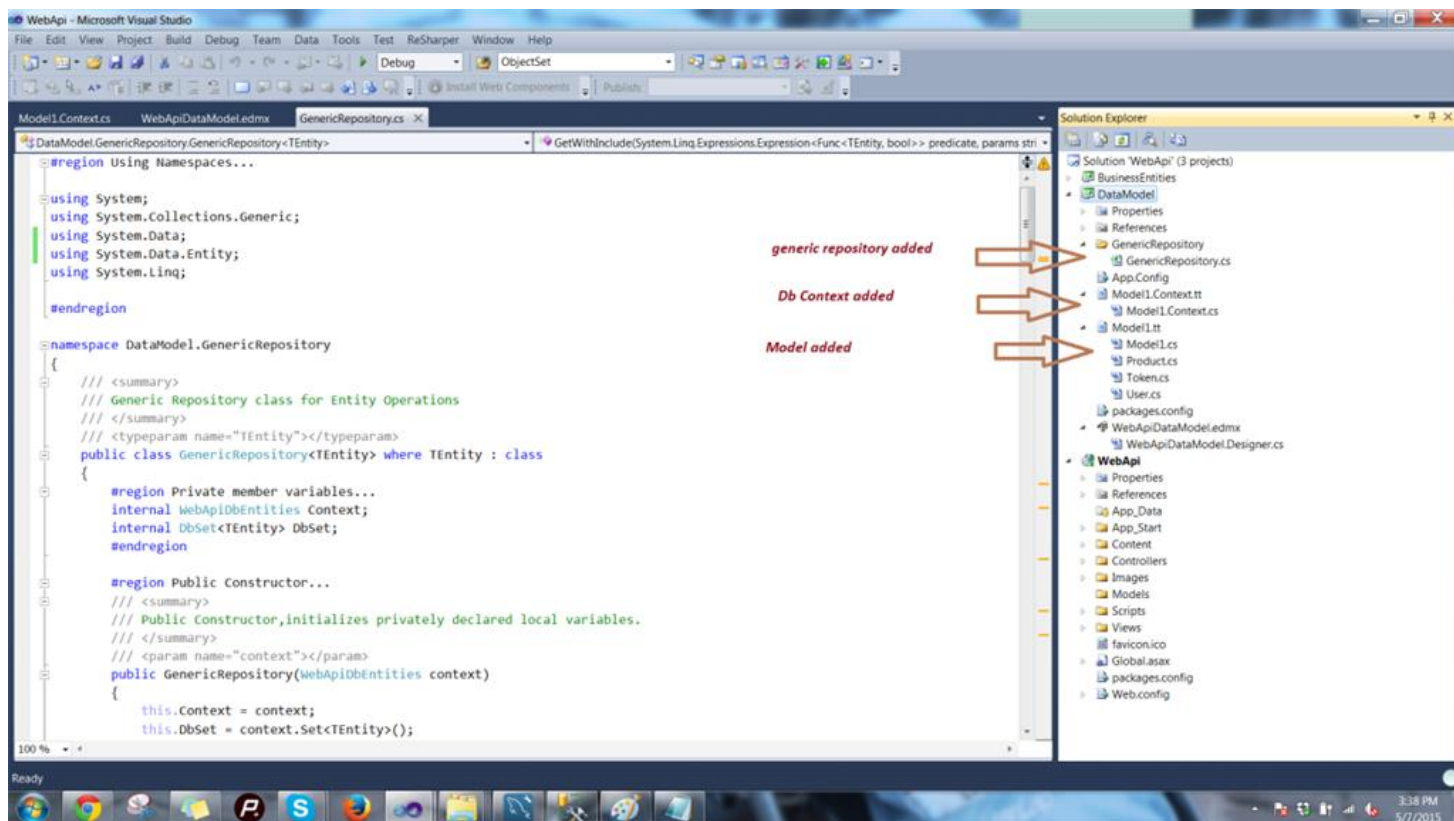


```

134.     }
135.     #endregion
136. }
137. }

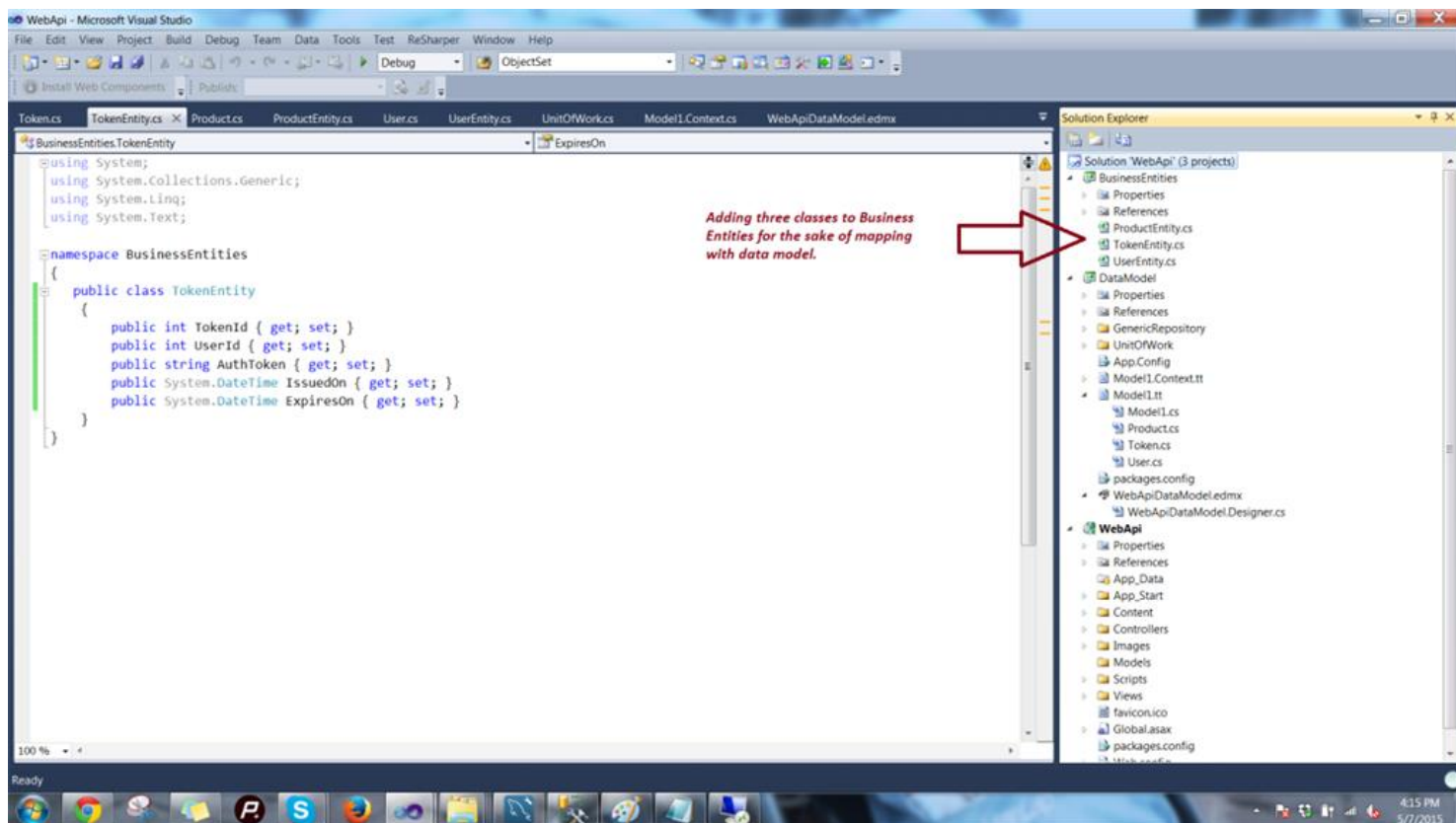
```

Now we have completely set up our data access layer and our project structure looks as shown below.



Setup Business Entities

Remember, we created a business entities project. You may wonder, since we already have database entities to interact with the database, why do we need Business Entities? The answer is as simple as, we are trying to follow a proper structure of communication and one would never want to expose the database entities to the end client, in our case the Web API, it involves a lot of risk. Hackers may manipulate the details and get access to your database. Instead, we'll use database entities in our business logic layer and use Business Entities as transfer objects to communicate between the business logic and the Web API project. So business entities may have different names, but their properties remain the same as database entities. In our case we'll add same-named business entity classes appended with the word "Entity" in our BusinessEntity project. So we'll end up having the following three classes:



Product Entity

1. **public class** ProductEntity
2. {
3. **public int** ProductId { **get; set;** }
4. **public string** ProductName { **get; set;** }
5. }

Token entity

1. **public class** TokenEntity
2. {
3. **public int** TokenId { **get; set;** }
4. **public int** UserId { **get; set;** }
5. **public string** AuthToken { **get; set;** }
6. **public** System.DateTime IssuedOn { **get; set;** }
7. **public** System.DateTime ExpiresOn { **get; set;** }
8. }

User Entity

1. **public class** UserEntity


```

2. {
3.     public int UserId { get; set; }
4.     public string UserName { get; set; }
5.     public string word { get; set; }
6.     public string Name { get; set; }
7. }

```

Setup Business Services Project

Add a new class library to the solution named BusinessServices. This layer will act as our business logic layer. Note that, we can use our API controllers to write the business logic, but I am trying to segregate business logic in an extra layer so that if in the future I want to use WCF, MVC, ASP.Net Web Pages or any other application as presentation layer then I can easily integrate Business logic layer into it.

We'll make this layer testable, so we need to create an interface and in it declare CRUD operations to be performed over a product table. Before we proceed, add a reference for the BusinessEntities project and DataModel project to this newly-created project.

Step 1

Create an interface named IProductServices and add the following code to it for CRUD operations methods.

```

1. using System.Collections.Generic;
2. using BusinessEntities;
3.
4. namespace BusinessServices
5. {
6.     /// <summary>
7.     /// Product Service Contract
8.     /// </summary>
9.     public interface IProductServices
10.    {
11.        ProductEntity GetProductById(int productId);
12.        IEnumerable<ProductEntity> GetAllProducts();
13.        int CreateProduct(ProductEntity productEntity);
14.        bool UpdateProduct(int productId, ProductEntity productEntity);
15.        bool DeleteProduct(int productId);

```



```
16. }  
17.}
```

Step 2

Create a class to implement this interface. Name that class ProductServices.

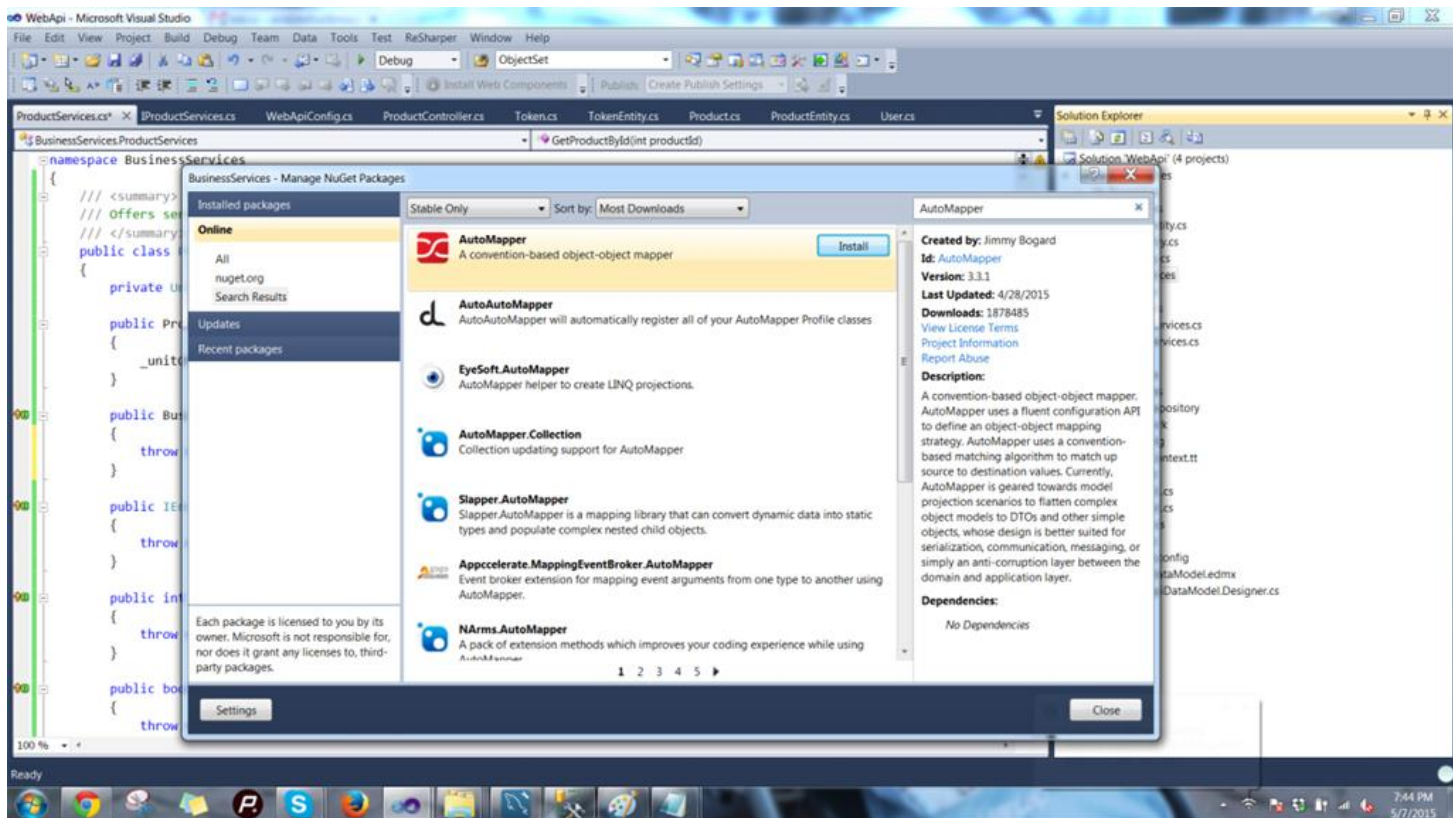
The class contains a private variable of UnitOfWork and a constructor to initialize that variable as in the following:

```
1. private readonly UnitOfWork _unitOfWork;  
2.  
3. /// <summary>  
4. /// Public constructor.  
5. /// </summary>  
6. public ProductServices()  
7. {  
8.     _unitOfWork = new UnitOfWork();  
9. }
```

We have decided not to expose our db entities to the Web API project, so we need something to map the db entities data to business entity classes. We'll make use of AutoMapper.

Step 3

Just right-click the project then select Extension manager, search for AutoMapper in the online gallery and add it to the BusinessServices project as in the following:



Step 4

Implement the following methods in the ProductServices class.

Add the following code to the class:

1. **using** System.Collections.Generic;
2. **using** System.Linq;
3. **using** System.Transactions;
4. **using** AutoMapper;
5. **using** BusinessEntities;
6. **using** DataModel;
7. **using** DataModel.UnitOfWork;
- 8.
9. **namespace** BusinessServices
10. {
11. **///** <summary>
12. **///** Offers services for product specific CRUD operations
13. **///** </summary>
14. **public class** ProductServices:IProductServices
15. {


```
16.     private readonly UnitOfWork _unitOfWork;
17.
18.     /// <summary>
19.     /// Public constructor.
20.     /// </summary>
21.     public ProductServices()
22.     {
23.         _unitOfWork = new UnitOfWork();
24.     }
25.
26.     /// <summary>
27.     /// Fetches product details by id
28.     /// </summary>
29.     /// <param name="productId"></param>
30.     /// <returns></returns>
31.     public BusinessEntities.ProductEntity GetProductById(int productId)
32.     {
33.         var product = _unitOfWork.ProductRepository.GetById(productId);
34.         if (product != null)
35.         {
36.             Mapper.CreateMap<Product, ProductEntity>();
37.             var productModel = Mapper.Map<Product, ProductEntity>(product);
38.             return productModel;
39.         }
40.         return null;
41.     }
42.
43.     /// <summary>
44.     /// Fetches all the products.
45.     /// </summary>
46.     /// <returns></returns>
47.     public IEnumerable<BusinessEntities.ProductEntity> GetAllProducts()
48.     {
49.         var products = _unitOfWork.ProductRepository.GetAll().ToList();
50.         if (products.Any())
51.         {
```



```
52.         Mapper.CreateMap<Product, ProductEntity>();
53.         var productsModel = Mapper.Map<List<Product>, List<ProductEntity>>(pro
ducts);
54.         return productsModel;
55.     }
56.     return null;
57. }
58.
59.     /// <summary>
60.     /// Creates a product
61.     /// </summary>
62.     /// <param name="productEntity"></param>
63.     /// <returns></returns>
64.     public int CreateProduct(BusinessEntities.ProductEntity productEntity)
65.     {
66.         using (var scope = new TransactionScope())
67.         {
68.             var product = new Product
69.             {
70.                 ProductName = productEntity.ProductName
71.             };
72.             _unitOfWork.ProductRepository.Insert(product);
73.             _unitOfWork.Save();
74.             scope.Complete();
75.             return product.ProductId;
76.         }
77.     }
78.
79.     /// <summary>
80.     /// Updates a product
81.     /// </summary>
82.     /// <param name="productId"></param>
83.     /// <param name="productEntity"></param>
84.     /// <returns></returns>
85.     public bool UpdateProduct(int productId, BusinessEntities.ProductEntity product
Entity)
```



```
86.     {
87.         var success = false;
88.         if (productEntity != null)
89.         {
90.             using (var scope = new TransactionScope())
91.             {
92.                 var product = _unitOfWork.ProductRepository.GetByID(productId);
93.                 if (product != null)
94.                 {
95.                     product.ProductName = productEntity.ProductName;
96.                     _unitOfWork.ProductRepository.Update(product);
97.                     _unitOfWork.Save();
98.                     scope.Complete();
99.                     success = true;
100.                }
101.            }
102.        }
103.        return success;
104.    }
105.
106.    /// <summary>
107.    /// Deletes a particular product
108.    /// </summary>
109.    /// <param name="productId"></param>
110.    /// <returns></returns>
111.    public bool DeleteProduct(int productId)
112.    {
113.        var success = false;
114.        if (productId > 0)
115.        {
116.            using (var scope = new TransactionScope())
117.            {
118.                var product = _unitOfWork.ProductRepository.GetByID(productId);
119.                if (product != null)
120.                {
121.
```



```

122.         _unitOfWork.ProductRepository.Delete(product);
123.         _unitOfWork.Save();
124.         scope.Complete();
125.         success = true;
126.     }
127. }
128. }
129.     return success;
130. }
131. }
132. }

```

Let me explain the idea of the code. We have the following 5 methods:

1. To get a product by id (GetproductById): We call the repository to get the product by id. Id is a parameter from the calling method to that service method. It returns the product entity from the database. Note that it will not return the exact db entity; instead we'll map it with our business entity using AutoMapper and return it to the calling method.

```

1. /// <summary>
2. /// Fetches product details by id
3. /// </summary>
4. /// <param name="productId"></param>
5. /// <returns></returns>
6. public BusinessEntities.ProductEntity GetProductById(int productId)
7. {
8.     var product = _unitOfWork.ProductRepository.GetByID(productId);
9.     if (product != null)
10.    {
11.        Mapper.CreateMap<Product, ProductEntity>();
12.        var productModel = Mapper.Map<Product, ProductEntity>(product);
13.        return productModel;
14.    }
15.    return null;
16.}

```


2. Get all products from the database (GetAllProducts): This method returns all the products residing in the database; again we use AutoMapper to map the list and return it.

```

1. /// <summary>
2. /// Fetches all the products.
3. /// </summary>
4. /// <returns></returns>
5. public IEnumerable<BusinessEntities.ProductEntity> GetAllProducts()
6. {
7.     var products = _unitOfWork.ProductRepository.GetAll().ToList();
8.     if (products.Any())
9.     {
10.         Mapper.CreateMap<Product, ProductEntity>();
11.         var productsModel = Mapper.Map<List<Product>, List<ProductEntity>>(products);
12.         return productsModel;
13.     }
14.     return null;
15. }

```

3. Create a new product (CreateProduct): This method takes the product's BusinessEntity as an argument and creates a new object of an actual database entity and inserts it using a unit of work.

```

1. /// <summary>
2. /// Creates a product
3. /// </summary>
4. /// <param name="productEntity"></param>
5. /// <returns></returns>
6. public int CreateProduct(BusinessEntities.ProductEntity productEntity)
7. {
8.     using (var scope = new TransactionScope())
9.     {
10.         var product = new Product
11.         {
12.             ProductName = productEntity.ProductName
13.         };
14.         _unitOfWork.ProductRepository.Insert(product);

```



```

15.     _unitOfWork.Save();
16.     scope.Complete();
17.     return product.ProductId;
18. }
19.}

```

I guess you can now write update and delete methods. So the following is the code for the complete class.

```

1. using System.Collections.Generic;
2. using System.Linq;
3. using System.Transactions;
4. using AutoMapper;
5. using BusinessEntities;
6. using DataModel;
7. using DataModel.UnitOfWork;
8.
9. namespace BusinessServices
10.{
11.    /// <summary>
12.    /// Offers services for product specific CRUD operations
13.    /// </summary>
14.    public class ProductServices:IProductServices
15.    {
16.        private readonly UnitOfWork _unitOfWork;
17.
18.        /// <summary>
19.        /// Public constructor.
20.        /// </summary>
21.        public ProductServices()
22.        {
23.            _unitOfWork = new UnitOfWork();
24.        }
25.
26.        /// <summary>

```



```
27.    /// Fetches product details by id
28.    /// </summary>
29.    /// <param name="productId"></param>
30.    /// <returns></returns>
31.    public BusinessEntities.ProductEntity GetProductById(int productId)
32.    {
33.        var product = _unitOfWork.ProductRepository.GetById(productId);
34.        if (product != null)
35.        {
36.            Mapper.CreateMap<Product, ProductEntity>();
37.            var productModel = Mapper.Map<Product, ProductEntity>(product);
38.            return productModel;
39.        }
40.        return null;
41.    }
42.
43.    /// <summary>
44.    /// Fetches all the products.
45.    /// </summary>
46.    /// <returns></returns>
47.    public IEnumerable<BusinessEntities.ProductEntity> GetAllProducts()
48.    {
49.        var products = _unitOfWork.ProductRepository.GetAll().ToList();
50.        if (products.Any())
51.        {
52.            Mapper.CreateMap<Product, ProductEntity>();
53.            var productsModel = Mapper.Map<List<Product>, List<ProductEntity>>(products);
54.            return productsModel;
55.        }
56.        return null;
57.    }
58.
59.    /// <summary>
60.    /// Creates a product
61.    /// </summary>
```



```
62.    /// <param name="productEntity"></param>
63.    /// <returns></returns>
64.    public int CreateProduct(BusinessEntities.ProductEntity productEntity)
65.    {
66.        using (var scope = new TransactionScope())
67.        {
68.            var product = new Product
69.            {
70.                ProductName = productEntity.ProductName
71.            };
72.            _unitOfWork.ProductRepository.Insert(product);
73.            _unitOfWork.Save();
74.            scope.Complete();
75.            return product.ProductId;
76.        }
77.    }
78.
79.    /// <summary>
80.    /// Updates a product
81.    /// </summary>
82.    /// <param name="productId"></param>
83.    /// <param name="productEntity"></param>
84.    /// <returns></returns>
85.    public bool UpdateProduct(int productId, BusinessEntities.ProductEntity product
    Entity)
86.    {
87.        var success = false;
88.        if (productEntity != null)
89.        {
90.            using (var scope = new TransactionScope())
91.            {
92.                var product = _unitOfWork.ProductRepository.GetById(productId);
93.                if (product != null)
94.                {
95.                    product.ProductName = productEntity.ProductName;
96.                    _unitOfWork.ProductRepository.Update(product);
```



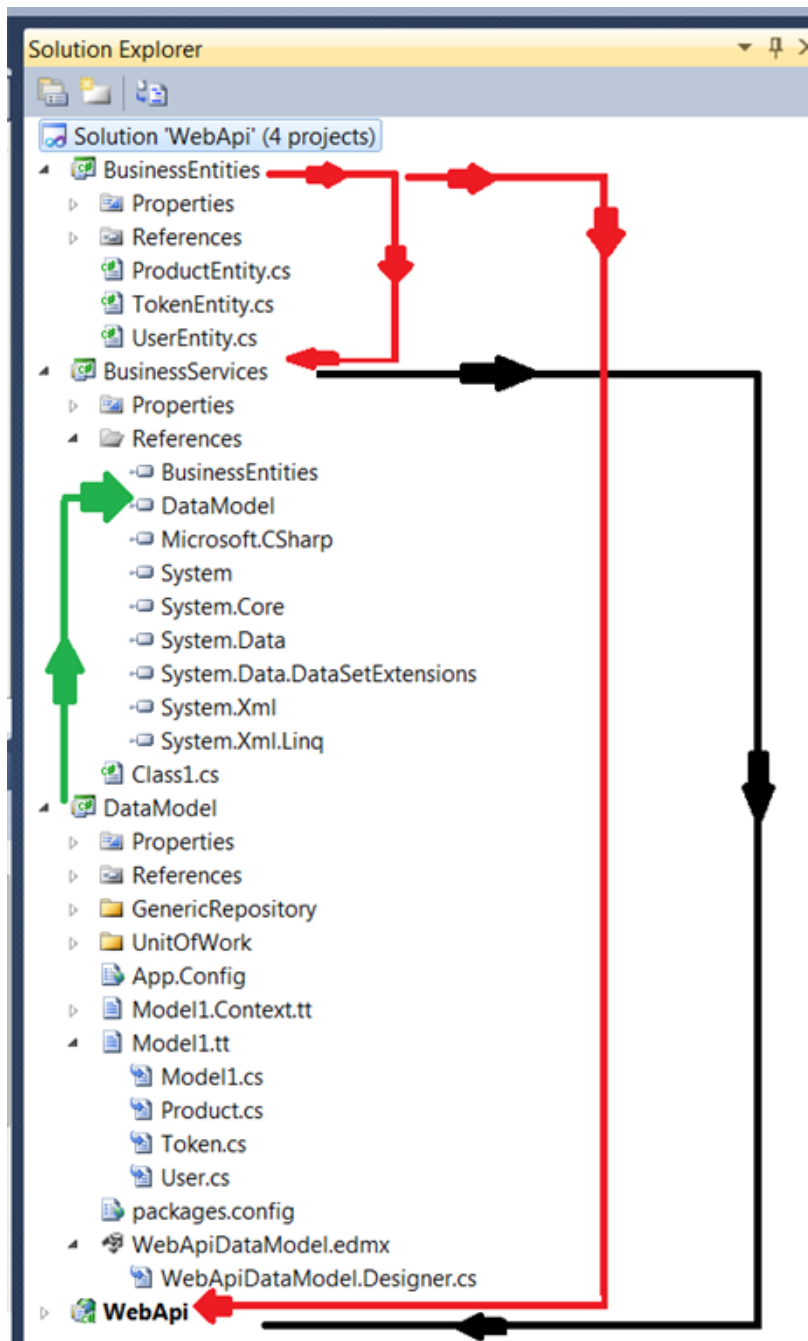
```
97.         _unitOfWork.Save();
98.         scope.Complete();
99.         success = true;
100.     }
101. }
102. }
103. return success;
104. }
105.
106. /// <summary>
107. /// Deletes a particular product
108. /// </summary>
109. /// <param name="productId"></param>
110. /// <returns></returns>
111. public bool DeleteProduct(int productId)
112. {
113.     var success = false;
114.     if (productId > 0)
115.     {
116.         using (var scope = new TransactionScope())
117.         {
118.             var product = _unitOfWork.ProductRepository.GetByID(productId);
119.             if (product != null)
120.             {
121.
122.                 _unitOfWork.ProductRepository.Delete(product);
123.                 _unitOfWork.Save();
124.                 scope.Complete();
125.                 success = true;
126.             }
127.         }
128.     }
129.     return success;
130. }
131. }
132. }
```


The job is now done at the business service level. Let's move on to the API controller to call these methods.

Setup WebAPI project

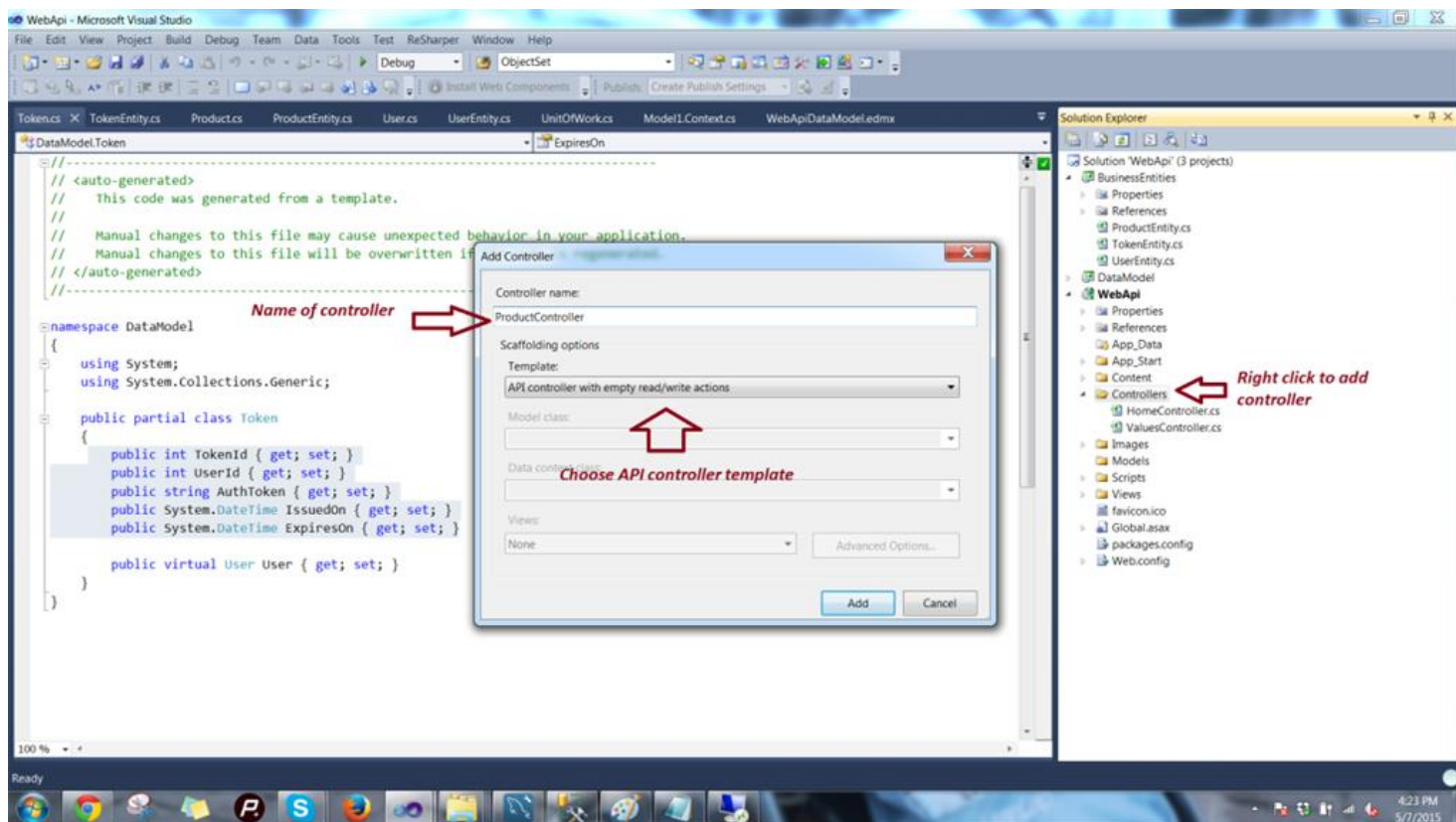
Step 1

Just add references for BusinessEntity and BusinessService to the WebAPI project, our architecture becomes like the following:



Step 2

Add a new WebAPI controller to the Controller folder. Right-click the Controller folder and add a new controller.



We get a controller as follows:

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Net;
5. **using** System.Net.Http;
6. **using** System.Web.Http;
- 7.
8. **namespace** WebApi.Controllers
9. {
10. **public class** ProductController : ApiController
11. {
12. **// GET api/product**
13. **public** IEnumerable<**string**> Get()
14. {
15. **return new string[]** { "value1", "value2" };
16. }
- 17.
18. **// GET api/product/5**


```
19.     public string Get(int id)
20.     {
21.         return "value";
22.     }
23.
24.     // POST api/product
25.     public void Post([FromBody]string value)
26.     {
27.     }
28.
29.     // PUT api/product/5
30.     public void Put(int id, [FromBody]string value)
31.     {
32.     }
33.
34.     // DELETE api/product/5
35.     public void Delete(int id)
36.     {
37.     }
38. }
39. }
```

We get HTTP VERBS as method names. The Web API is smart enough to recognize a request with the name of the VERB itself. In our case we are doing CRUD operations, so we don't need to change the names of the method, we just needed this. We only need to write the calling logic inside these methods. In further chapters of the book, we will work out how to define new routes and provide method names of our choice with those routes.

Step 3

Add logic to call Business Service methods. Just make an object of the Business Service and call its respective methods, our Controller class becomes like:

1. **using** System.Collections.Generic;
2. **using** System.Linq;
3. **using** System.Net;


```

4. using System.Net.Http;
5. using System.Web.Http;
6. using BusinessEntities;
7. using BusinessServices;
8.
9. namespace WebApi.Controllers
10. {
11.     public class ProductController : ApiController
12.     {
13.
14.         private readonly IProductServices _productServices;
15.
16.         #region Public Constructor
17.
18.         /// <summary>
19.         /// Public constructor to initialize product service instance
20.         /// </summary>
21.         public ProductController()
22.         {
23.             _productServices = new ProductServices();
24.         }
25.
26.         #endregion
27.
28.         // GET api/product
29.         public HttpResponseMessage Get()
30.         {
31.             var products = _productServices.GetAllProducts();
32.             if(products!=null)
33.             {
34.                 var productEntities = products as List<ProductEntity> ?? products.ToList();
35.                 if (productEntities.Any())
36.                     return Request.CreateResponse(HttpStatusCode.OK, productEntities);
37.             }
38.             return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not found");

```

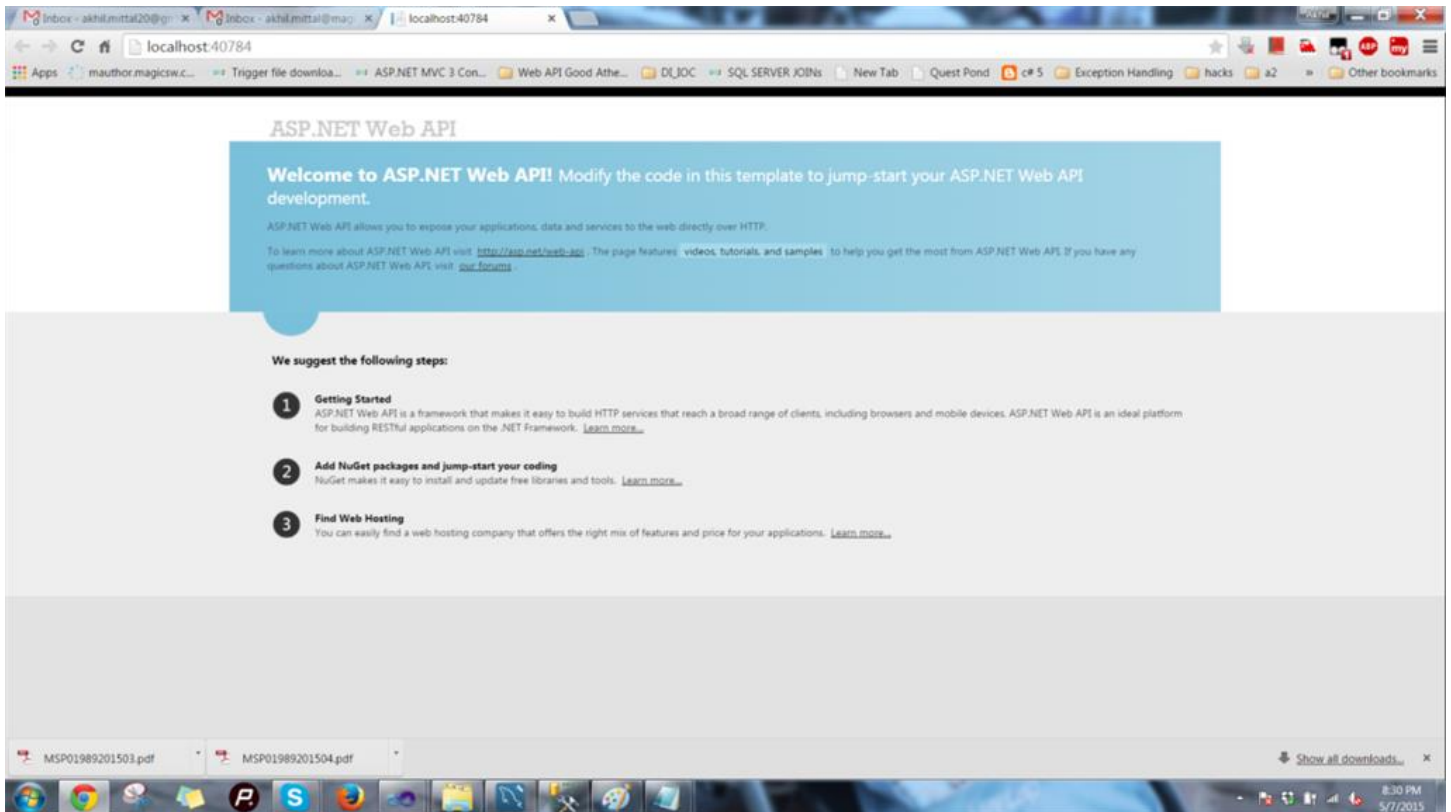


```
39.     }
40.
41.     // GET api/product/5
42.     public HttpResponseMessage Get(int id)
43.     {
44.         var product = _productServices.GetProductById(id);
45.         if (product != null)
46.             return Request.CreateResponse(HttpStatusCode.OK, product);
47.         return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product
    found for this id");
48.     }
49.
50.     // POST api/product
51.     public int Post([FromBody] ProductEntity productEntity)
52.     {
53.         return _productServices.CreateProduct(productEntity);
54.     }
55.
56.     // PUT api/product/5
57.     public bool Put(int id, [FromBody]ProductEntity productEntity)
58.     {
59.         if (id > 0)
60.         {
61.             return _productServices.UpdateProduct(id, productEntity);
62.         }
63.         return false;
64.     }
65.
66.     // DELETE api/product/5
67.     public bool Delete(int id)
68.     {
69.         if (id > 0)
70.             return _productServices.DeleteProduct(id);
71.         return false;
72.     }
73. }
```



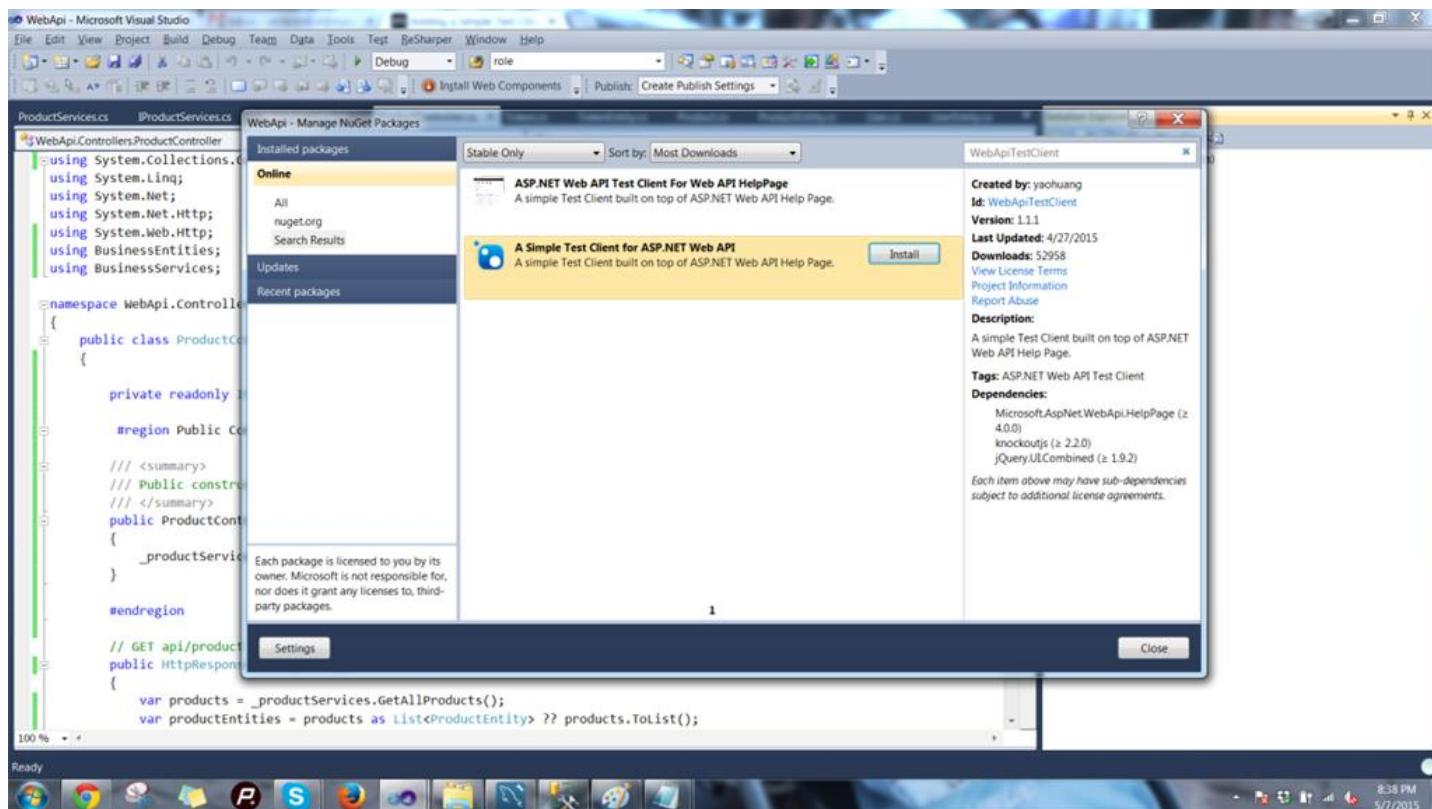
```
74.}
```

Just run the application and we will get:

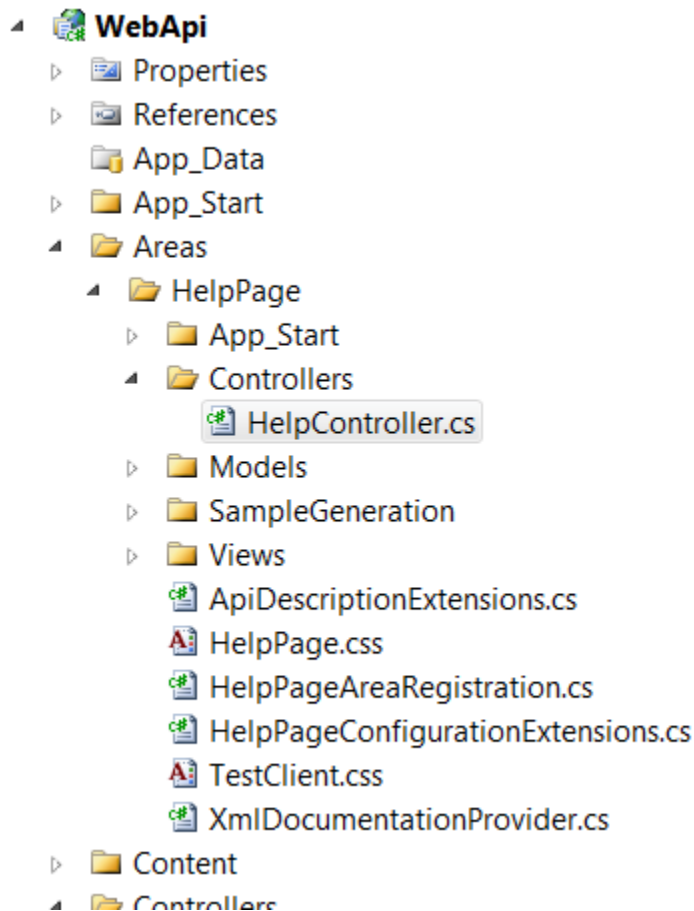


But now how do we test our API? We don't have a client. Guys, we'll not be writing a client now to test it. We'll add a package that will do all the work.

Just go to Manage Nuget Packages by right-clicking WebAPI project and type WebAPITestClient into the search box in online packages as in the following:



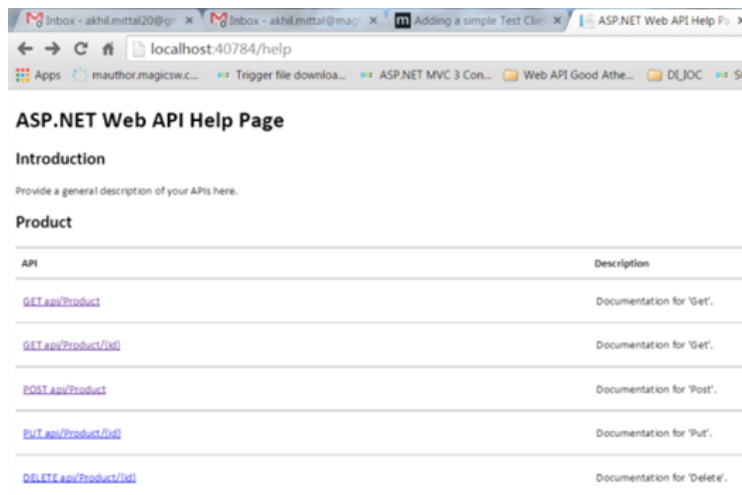
You'll get "A simple Test Client for ASP.NET Web API", just add it. You'll get a help controller in Areas -> HelpPage like shown below.



Running the Application

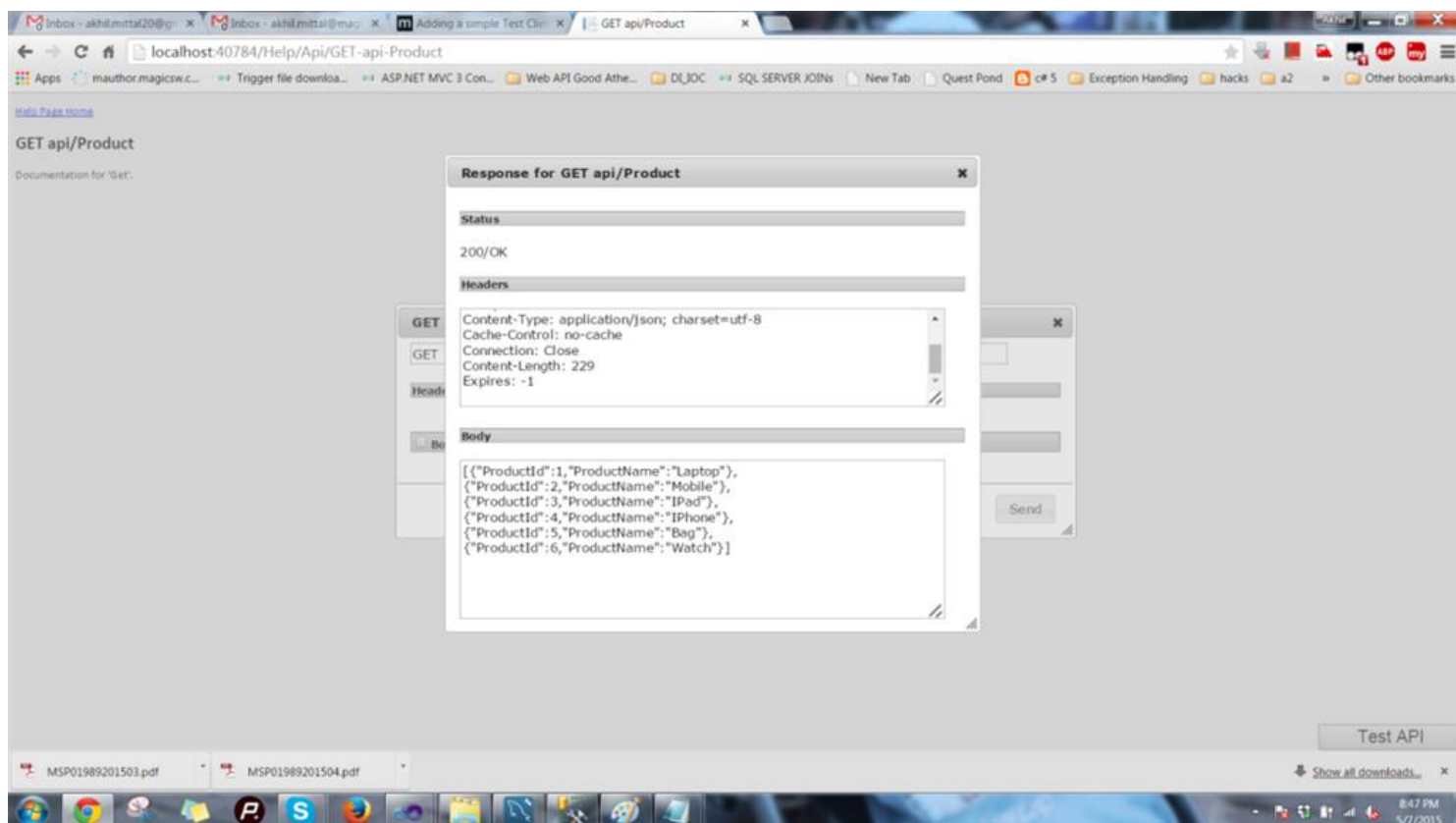
Before running the application, I have put some test data in our product table.

Just hit F5, you get the same page as you got earlier, just append “/help” to its URL and you'll get the test client as in the following:

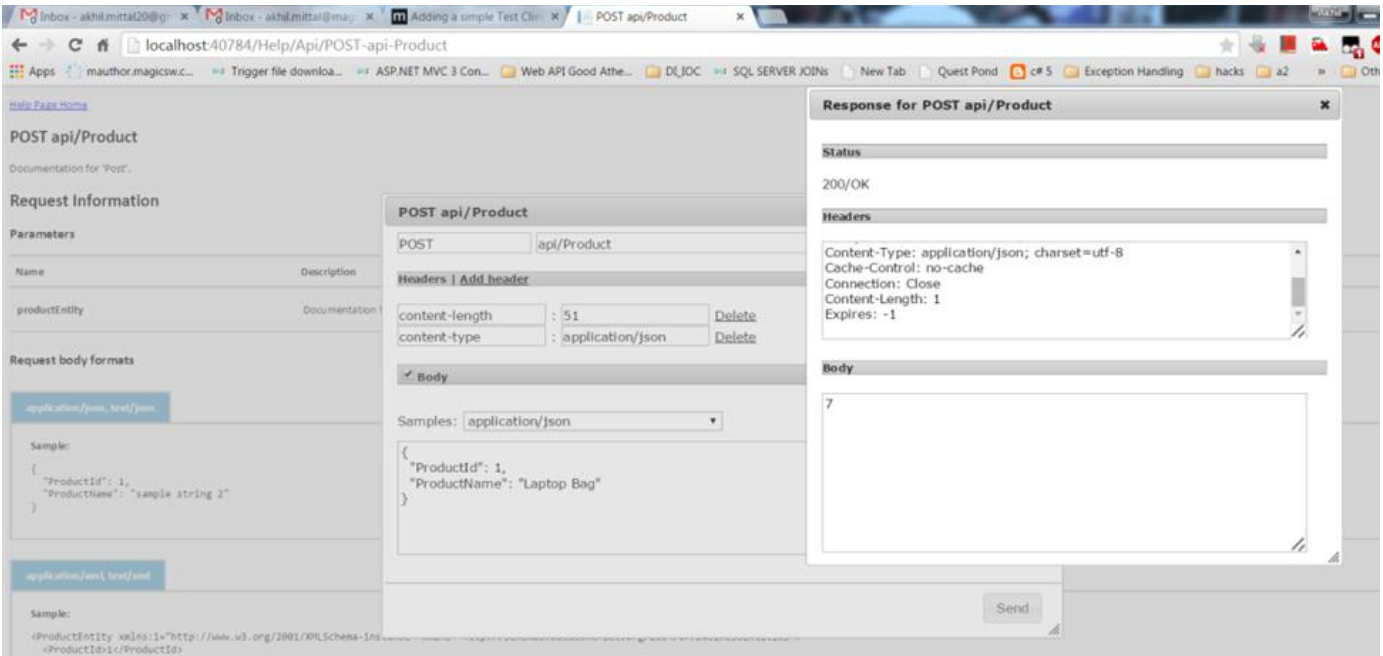


You can test each service by clicking on it.

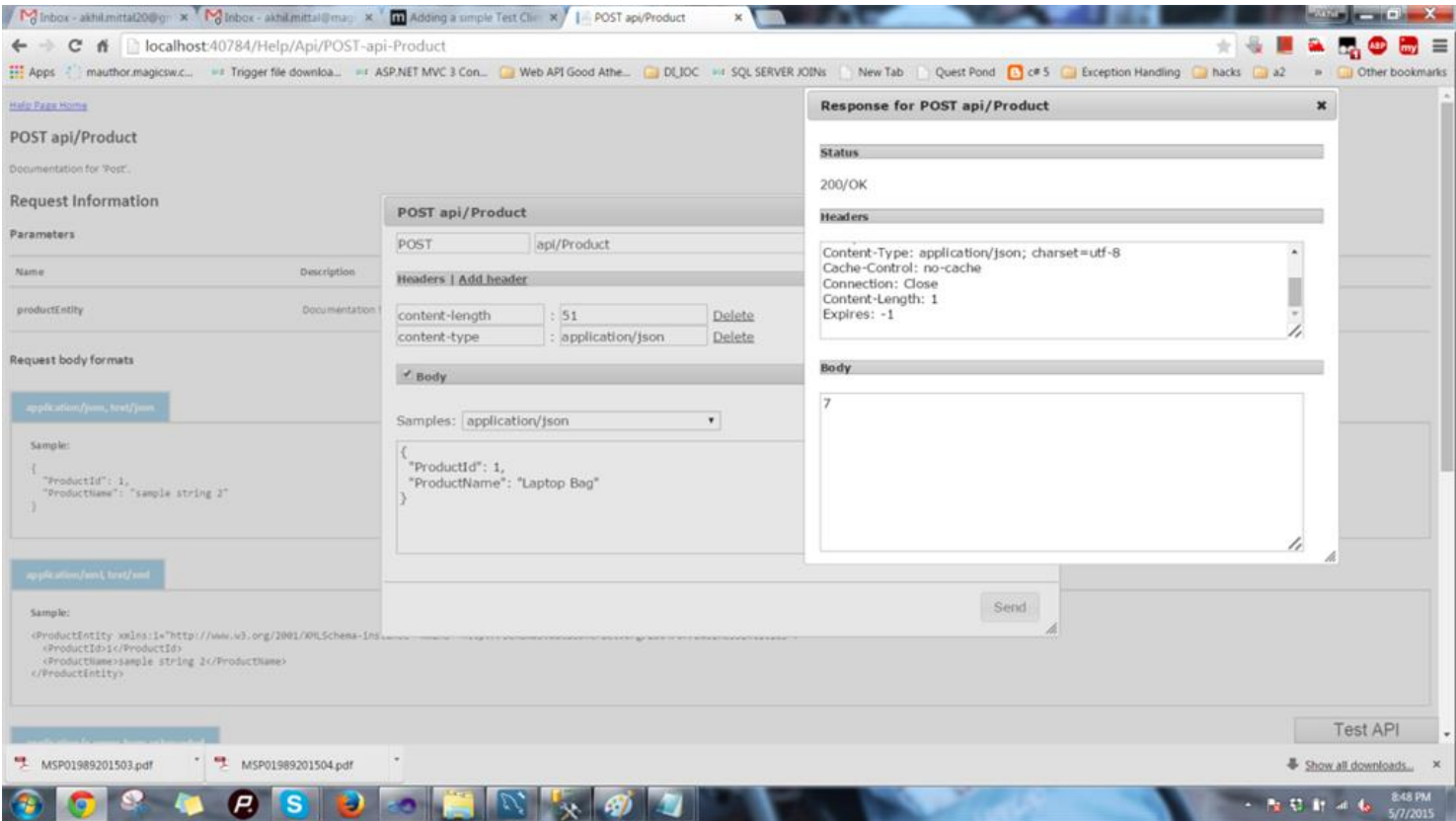
Service for GetAllProduct:



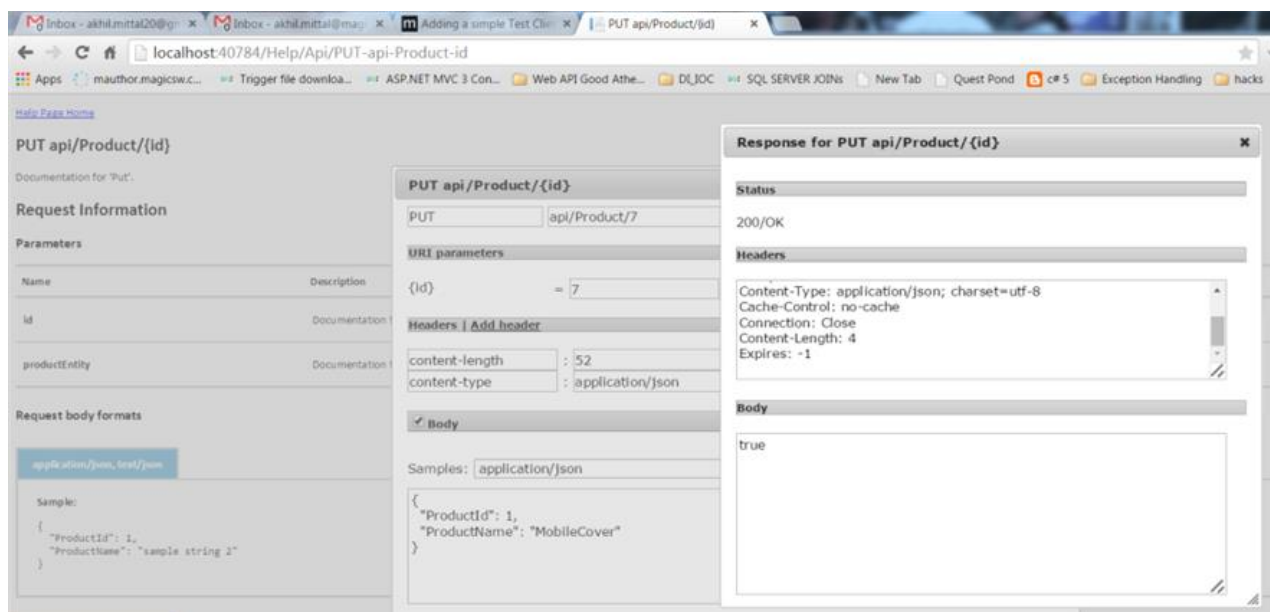
To create a new product:



In the database, we get a new product as in the following:



Update product:



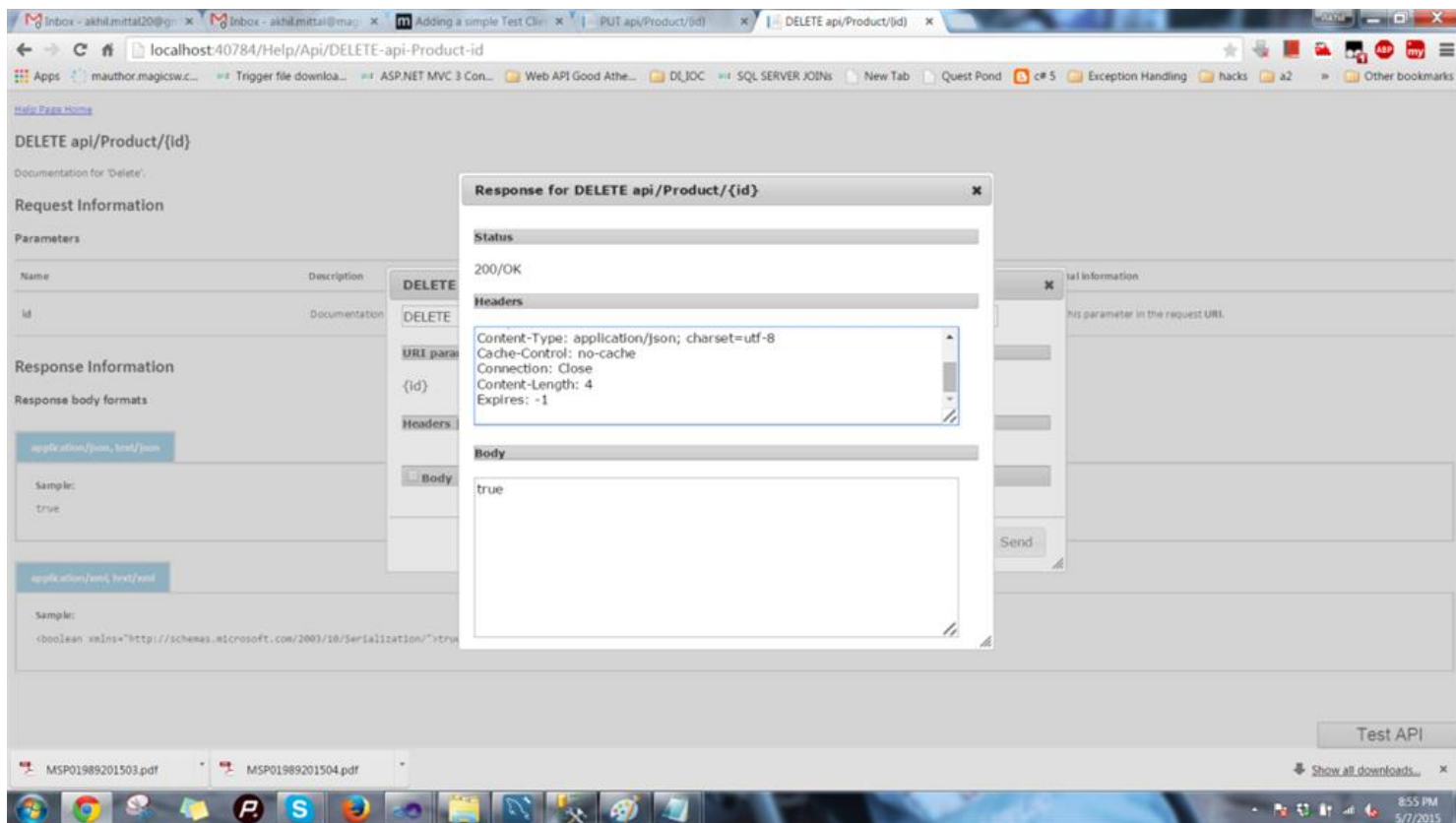
We get the following in the database:

69B5ZR1.WebApiDb - dbo.Products		
ProductId	ProductName	
1	Laptop	
2	Mobile	
3	IPad	
4	IPhone	
5	Bag	
6	Watch	
7	MobileCover	
*	NULL	

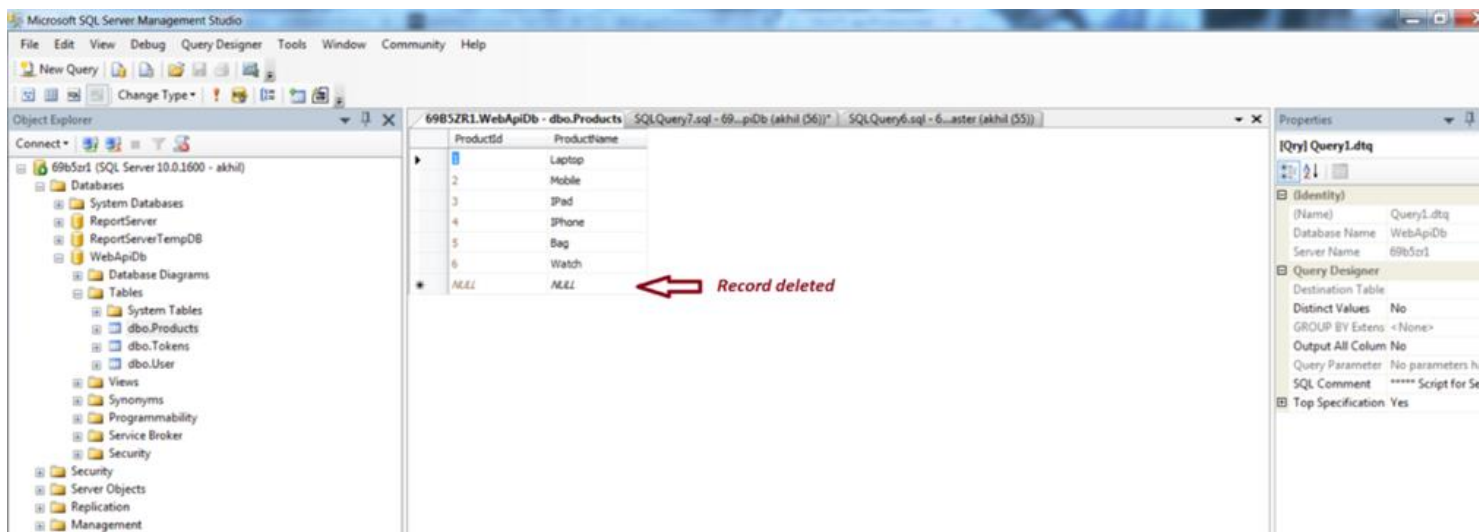
SQLQuery7.sql - 69...piDb (akhil (56))* SQLC

Updated record

Delete product:



In the database:



Job done.

Design Flaws

1. The architecture is tightly coupled. Inversion of Control (IOC) needs to be there.
2. We cannot define our own routes.
3. No exception handling and logging.
4. No unit tests.

Conclusion

We now know how to create a WebAPI and perform CRUD operations using an n layered architecture.

But still, there are some flaws in this design. In next chapter I'll explain how to make the system loosely-coupled using the Dependency Injection Principle. We'll also cover all the design flaws to make our design better and stronger. Until then Happy Coding.

Source Code

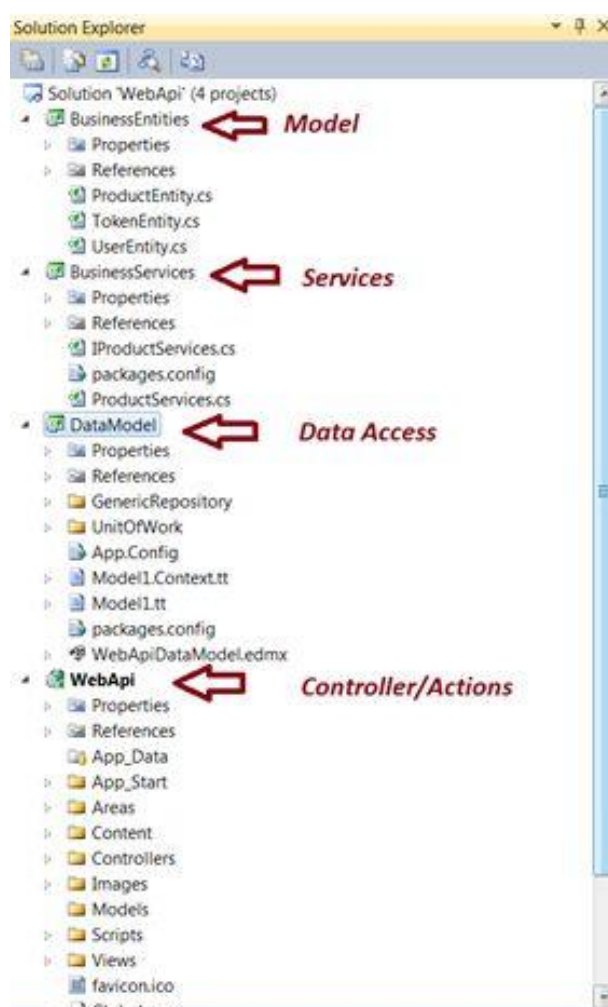
You can also download the source code from [GitHub](#).

Inversion of Control Using Dependency Injection in Web API's Using Unity Container and Bootstrapper

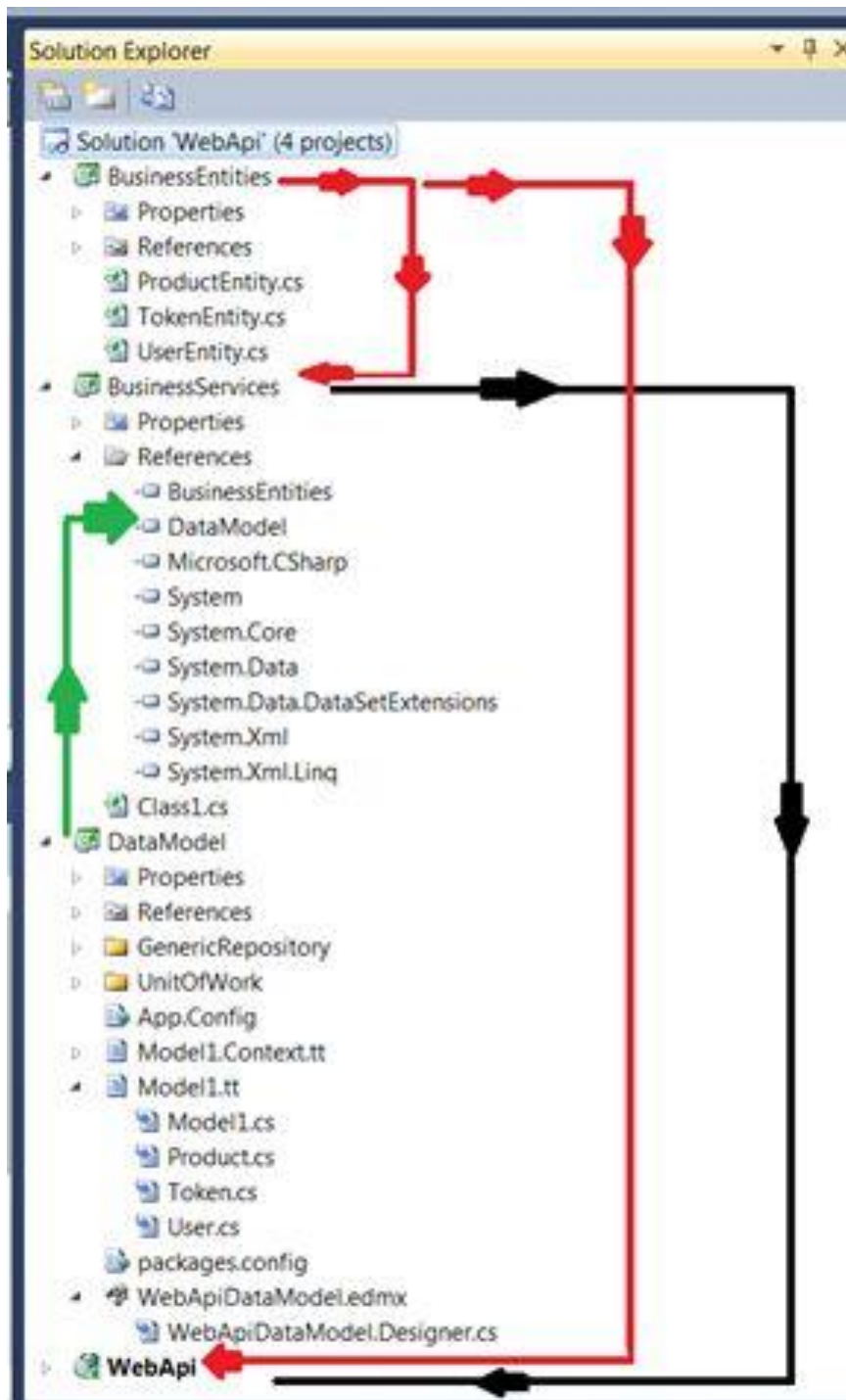
This chapter explains how to make our Web API service architecture loosely coupled and more flexible. We already learned how to create a RESTful service using the ASP.Net Web API and Entity framework. If you remember, we ended up with a solution with a design flaw; we'll try to overcome that flaw by resolving the dependencies of dependent components. There are various methods you can use to resolve dependency of components. In this chapter I'll explain how to resolve a dependency using Unity Container provided by Microsoft's Unity Application Block.

Existing Design and Problem

We already have an existing design. If you open the solution, you'll get to see the structure as shown below.



The modules are dependent in a way,



There is no problem with the structure, but the way they interact with each other is really problematic. You must have noticed that we are trying to communicate with layers, making the physical objects of classes.

For example, the Controller's constructor makes an object of the Service layer to communicate as in the following:


```
1. /// <summary>
2. /// Public constructor to initialize product service instance
3. /// </summary>
4. public ProductController()
5. {
6.     _productServices = new ProductServices();
7. }
```

The Service constructor in turn makes an object of UnitOfWork to communicate with the database as in the following:

```
1. /// <summary>
2. /// Public constructor.
3. /// </summary>
4. public ProductServices()
5. {
6.     _unitOfWork = new UnitOfWork();
7. }
```

The problem lies in these portions of code. We see the Controller is dependent upon instantiation of the service and the service is dependent upon UnitOfWork to be instantiated. Our layers should not be that tightly coupled and should be dependant on each other.

The work of creating an object should be assigned to someone else. Our layers should not worry about the creation of objects.

We'll assign this role to a third-party that will be called our container. Fortunately Unity provides that help to us, to eliminate this dependency problem and invert the control flow by injecting a dependency, not by creating objects but by using constructors or properties. There are other methods too, but I am not going into detail.

Introduction to Unity

You can have a read about Unity from [this](#) link, I am just quoting some lines.

“The Unity Application Block (Unity) is a lightweight, extensible dependency injection container that supports constructor injection, property injection and method call injection. It provides developers with the following advantages:

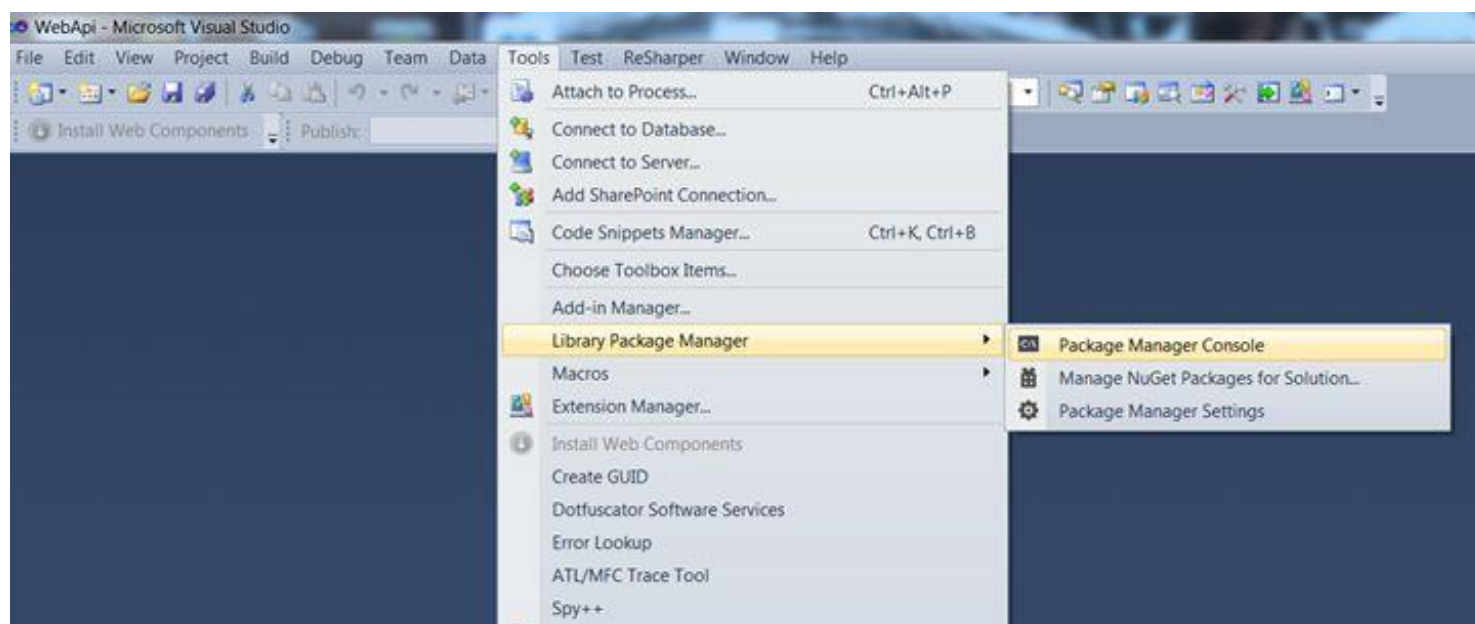
- It provides simplified object creation, especially for hierarchical object structures and dependencies that simplifies application code.
- It supports abstraction of requirements; this allows developers to specify dependencies at run time or in configuration and simplify management of crosscutting concerns.
- It increases flexibility by deferring component configuration to the container.
- It has a service location capability; this allows clients to store or cache the container. This is especially useful in ASP.NET Web applications where developers can persist the container in the ASP.NET session or application.”

Setup Unity

Open your Visual Studio, I am using Visual Studio 2010, you can use Visual Studio version 2010 or above. Load the solution.

Step 1

Browse to Tools -> Library Packet Manager - > Packet Manager Console as in the following:



We'll add a package for a Unity Application Block.

In the bottom-left corner of Visual Studio, you'll find where to write the command.

Type command `Unity.MVC3` and choose “WebApi” project before you fire the command.



Step 2

Now for the Bootstrapper class.

Unity.Mvc3 comes with a Bootstrapper class. As soon as you run the command, the Bootstrapper class will be generated in your WebAPI project as in the following:

```

1. using System.Web.Http;
2. using System.Web.Mvc;
3. using BusinessServices;
4. using DataModel.UnitOfWork;
5. using Microsoft.Practices.Unity;
6. using Unity.Mvc3;
7.
8. namespace WebApi
9. {
10.     public static class Bootstrapper
11.     {
12.         public static void Initialise()
13.         {
14.             var container = BuildUnityContainer();
15.
16.             DependencyResolver.SetResolver(new UnityDependencyResolver(container));
17.
18.             // register dependency resolver for WebAPI RC
19.             GlobalConfiguration.Configuration.DependencyResolver = new Unity.WebApi.
                UnityDependencyResolver(container);
20.         }
21.
22.         private static IUnityContainer BuildUnityContainer()

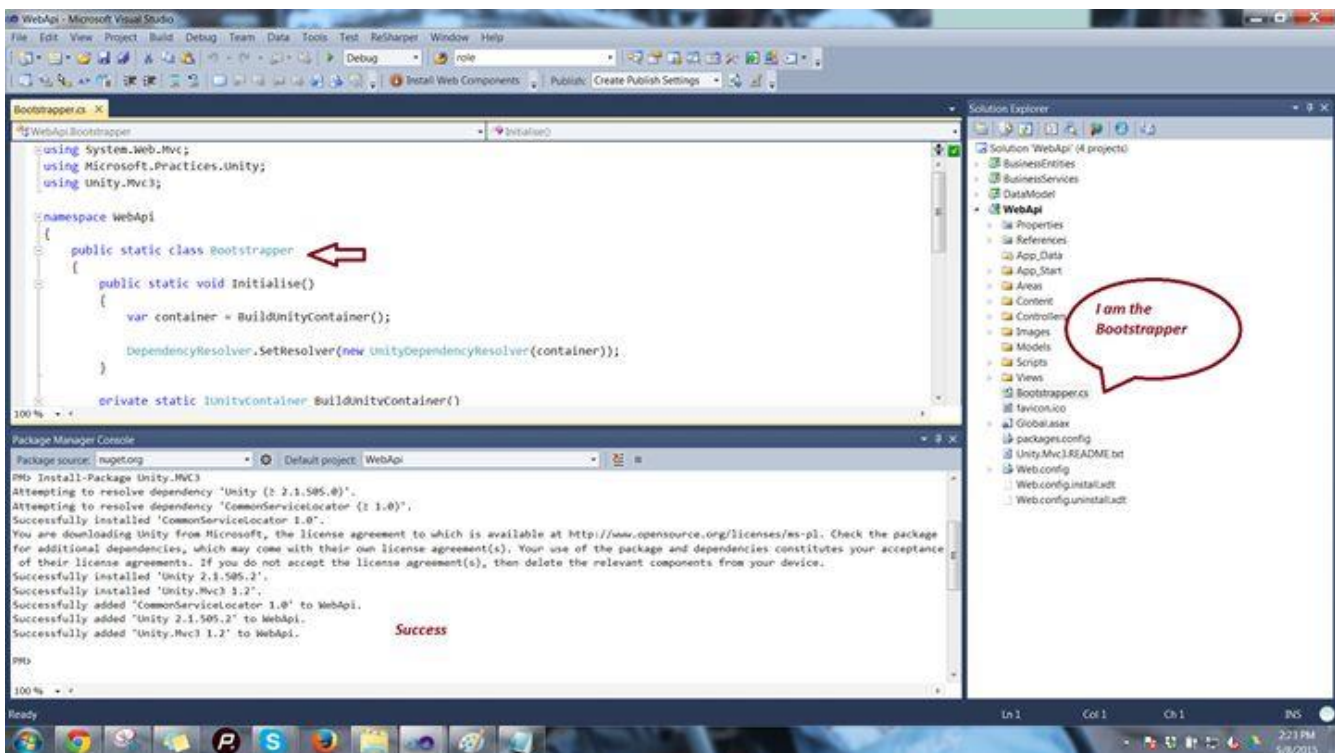
```



```

23.  {
24.      var container = new UnityContainer();
25.
26.      // register all your components with the container here
27.      // it is NOT necessary to register your controllers
28.
29.      // e.g. container.RegisterType<ITestService, TestService>();
30.      container.RegisterType<IProductServices, ProductServices>().RegisterType<Un
itOfWork>(new HierarchicalLifetimeManager());
31.
32.      return container;
33.  }
34. }
35.}

```



This class comes with an initial configuration to set up your container. All the functionality is builtin, we only need to specify the dependencies that we need to resolve in the “BuildUnityContainer”, like it says in the commented statement as in the following:

1. // register all your components with the container here
2. // it is NOT necessary to register your controllers
- 3.

4. `// e.g. container.RegisterType<ITestService, TestService>();`

Step 3

Just specify the components below these commented lines that we need to resolve. In our case, it's ProductServices and UnitOfWork, so just add:

1. `container.RegisterType<IProductServices, ProductServices>().RegisterType<UnitOfWork>(new HierarchicalLifetimeManager());`

"HierarchicalLifetimeManager" , for this lifetime manager, as for the ContainerControlledLifetimeManager, Unity returns the same instance of the registered type or object each time you call the Resolve or ResolveAll method or when the dependency mechanism injects instances into other classes. The distinction is that when there are child containers, each child resolves its own instance of the object and does not share one with the parent. When resolving in the parent, the behavior is like a container controlled lifetime; when resolving the parent and the child you have different instances with each acting as a container-controlled lifetime. If you have multiple children, each will resolve its own instance.

If you don't find "UnitOfWork", just add a reference to the DataModel project in the WebAPI project.

So our Bootstrapper class becomes:

```
1. public static class Bootstrapper
2. {
3.     public static void Initialise()
4.     {
5.         var container = BuildUnityContainer();
6.
7.         DependencyResolver.SetResolver(new UnityDependencyResolver(container));
8.
9.         // register dependency resolver for WebAPI RC
10.        GlobalConfiguration.Configuration.DependencyResolver = new Unity.WebApi.UnityDependencyResolver(container);
11.    }
```



```

12.
13. private static IUnityContainer BuildUnityContainer()
14. {
15.     var container = new UnityContainer();
16.
17.     // register all your components with the container here
18.     // it is NOT necessary to register your controllers
19.
20.     // e.g. container.RegisterType<ITestService, TestService>();
21.     container.RegisterType<IProductServices, ProductServices>().RegisterType<Unit
        OfWork>(new HierarchicalLifetimeManager());
22.
23.     return container;
24. }
25.}

```

Like that we can also specify other dependent objects in BuildUnityContainerMethod.

Step 4

Now we need to call the Initialise method of the Bootstrapper class. Note, we need the objects as soon as our modules load, therefore we require the container to do its work when the application loads, therefore go to the Global.asax file and add one line to call the Initialise method. Since this is a static method, we can directly call it using the class name as in the following:

```
1. Bootstrapper.Initialise();
```

Our global.asax becomes,

```

1. using System.Linq;
2. using System.Web.Http;
3. using System.Web.Mvc;
4. using System.Web.Optimization;
5. using System.Web.Routing;
6. using Newtonsoft.Json;
7. using WebApi.App_Start;
8.
9. namespace WebApi

```



```

10.{
11.  // Note: For instructions on enabling IIS6 or IIS7 classic mode,
12.  // visit http://go.microsoft.com/?LinkId=9394801
13.
14.  public class WebApiApplication : System.Web.HttpApplication
15.  {
16.      protected void Application_Start()
17.      {
18.          AreaRegistration.RegisterAllAreas();
19.
20.          WebApiConfig.Register(GlobalConfiguration.Configuration);
21.          FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
22.          RouteConfig.RegisterRoutes(RouteTable.Routes);
23.          BundleConfig.RegisterBundles(BundleTable.Bundles);
24.          //Initialise Bootstrapper
25.          Bootstrapper.Initialise();
26.
27.          //Define Formatters
28.          var formatters = GlobalConfiguration.Configuration.Formatters;
29.          var jsonFormatter = formatters.JsonFormatter;
30.          var settings = jsonFormatter.SerializerSettings;
31.          settings.Formatting = Formatting.Indented;
32.          // settings.ContractResolver = new CamelCasePropertyNamesContractResolve
33.          var appXmlType = formatters.XmlFormatter.SupportedMediaTypes.FirstOrDefault(t => t.MediaType == "application/xml");
34.          formatters.XmlFormatter.SupportedMediaTypes.Remove(appXmlType);
35.
36.          //Add CORS Handler
37.          GlobalConfiguration.Configuration.MessageHandlers.Add(new CorsHandler());
38.      }
39.  }
40.}

```

Half of the job is done. We now need to touch base with our controller and service class constructors to utilize the instances already created for them at application load.

Setup Controller

We have already set up Unity in our application. There are various methods in which we can inject a dependency, like constructor injection, property injection, via a service locator. I am here using Constructor Injection, because I find it the best method to use with Unity Container to resolve dependency.

Just go to your ProductController, you find your constructor written as:

```
1. /// <summary>
2. /// Public constructor to initialize product service instance
3. /// </summary>
4. public ProductController()
5. {
6.     _productServices = new ProductServices();
7. }
```

Just add a parameter to your constructor that takes your ProductServices reference, as we did in the following:

```
1. /// <summary>
2. /// Public constructor to initialize product service instance
3. /// </summary>
4. public ProductController(IProductServices productServices)
5. {
6.     _productServices = productServices;
7. }
```

And initialize your “productServices” variable with the parameter. In this case, when the constructor of the controller is called, it will be served with a pre-instantiated service instance and does not need to create an instance of the service, our Unity container did the job of object creation.

Setup Services

For services too, we proceed in a similar fashion. Just open your ProductServices class, we see the dependency of UnitOfWork here as in the following:

```
1. /// <summary>
2. /// Public constructor.
```



```
3. /// </summary>
4. public ProductServices()
5. {
6.     _unitOfWork = new UnitOfWork();
7. }
```

Again, we perform the same steps and a parameter of type `UnitOfWork` to our constructor.

Our code becomes:

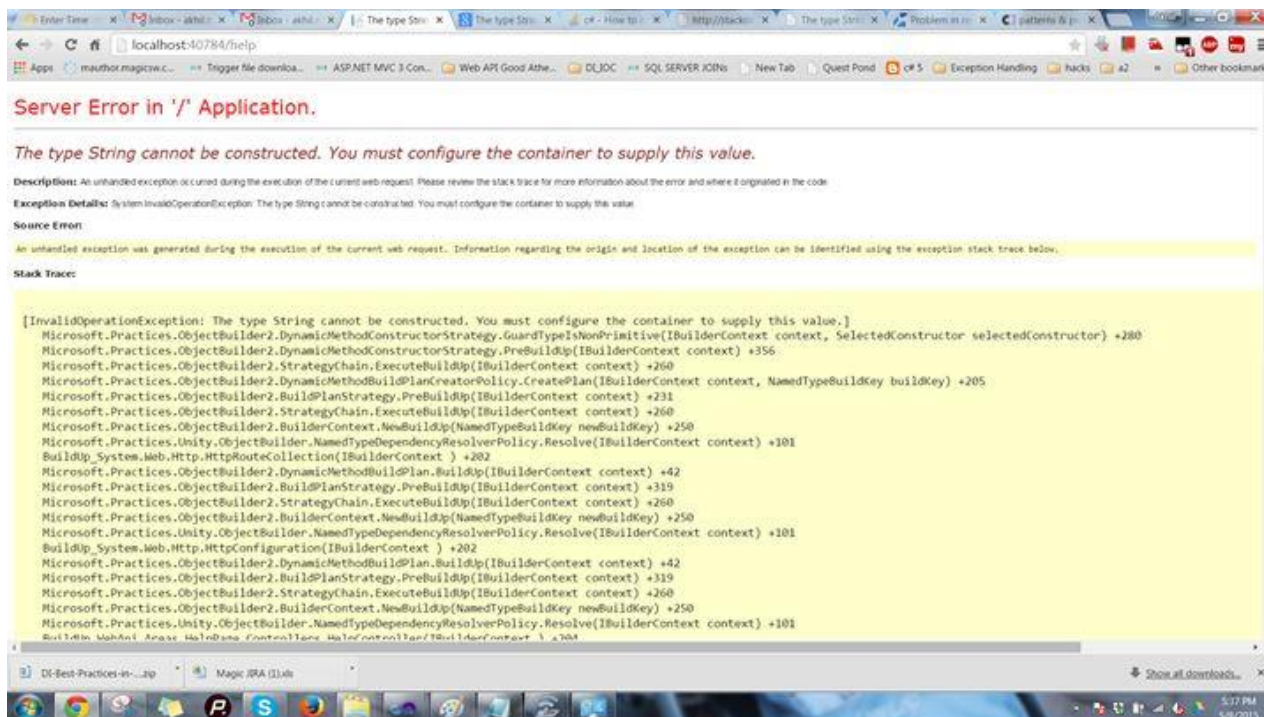
```
1. /// <summary>
2. /// Public constructor.
3. /// </summary>
4. public ProductServices(UnitOfWork unitOfWork)
5. {
6.     _unitOfWork = unitOfWork;
7. }
```

Here we'll also get the pre-instantiated object on `UnitOfWork`. So the service does not need to worry about creating objects. Remember, we did `.RegisterType<UnitOfWork>()` in the `Bootstrapper` class.

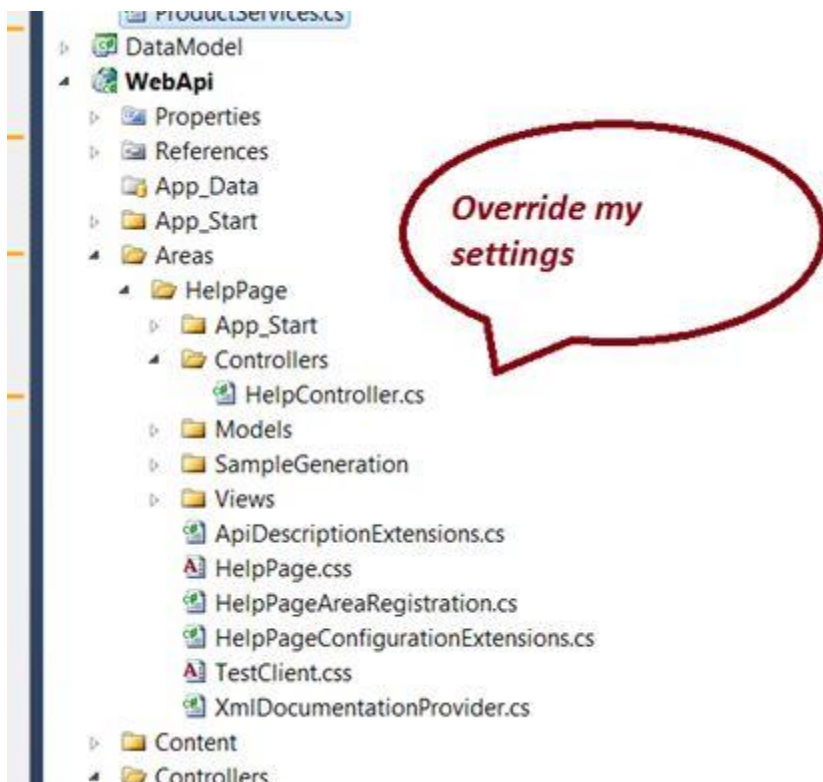
We have now made our components independent.

Running the application

Our job is nearly done. We need to run the application. Just hit F5. To our surprise we'll end up with in an error page as in the following:



Do you remember? We added a test client to our project to test our API in first chapter. That test client has a controller too, we need to override its settings to make our application work. Just go to Areas -> HelpPage -> Controllers -> HelpController in the WebAPI project as in the following:



Comment out the existing constructors and add a Configuration property like shown below.


```

1. //Remove constructors and existing Configuration property.
2.
3. //public HelpController()
4. //    : this(GlobalConfiguration.Configuration)
5. //{
6. //}
7.
8. //public HelpController(HttpConfiguration config)
9. //{
10.//    Configuration = config;
11.//}
12.
13.//public HttpConfiguration Configuration { get; private set; }
14.
15./// <summary>
16./// Add new Configuration Property
17./// </summary>
18.protected static HttpConfiguration Configuration
19.{
20.    get { return GlobalConfiguration.Configuration; }
21.}

```

Our controller code becomes:

```

1. using System;
2. using System.Web.Http;
3. using System.Web.Mvc;
4. using WebApi.Areas.HelpPage.Models;
5.
6. namespace WebApi.Areas.HelpPage.Controllers
7. {
8.     /// <summary>
9.     /// The controller that will handle requests for the help page.
10.    /// </summary>
11.    public class HelpController : Controller
12.    {
13.        //Remove constructors and existing Configuration property.

```



```
14.
15.     //public HelpController()
16.     //    : this(GlobalConfiguration.Configuration)
17.     //{
18.     //}
19.
20.     //public HelpController(HttpConfiguration config)
21.     //{
22.     //    Configuration = config;
23.     //}
24.
25.     //public HttpConfiguration Configuration { get; private set; }
26.
27.     /// <summary>
28.     /// Add new Configuration Property
29.     /// </summary>
30.     protected static HttpConfiguration Configuration
31.     {
32.         get { return GlobalConfiguration.Configuration; }
33.     }
34.
35.     public ActionResult Index()
36.     {
37.         return View(Configuration.Services.GetApiExplorer().ApiDescriptions);
38.     }
39.
40.     public ActionResult Api(string apild)
41.     {
42.         if (!String.IsNullOrEmpty(apild))
43.         {
44.             HelpPageApiModel apiModel = Configuration.GetHelpPageApiModel(apild);
45.
46.             if (apiModel != null)
47.             {
48.                 return View(apiModel);
49.             }
```



```

49.     }
50.
51.     return View("Error");
52. }
53. }
54.}

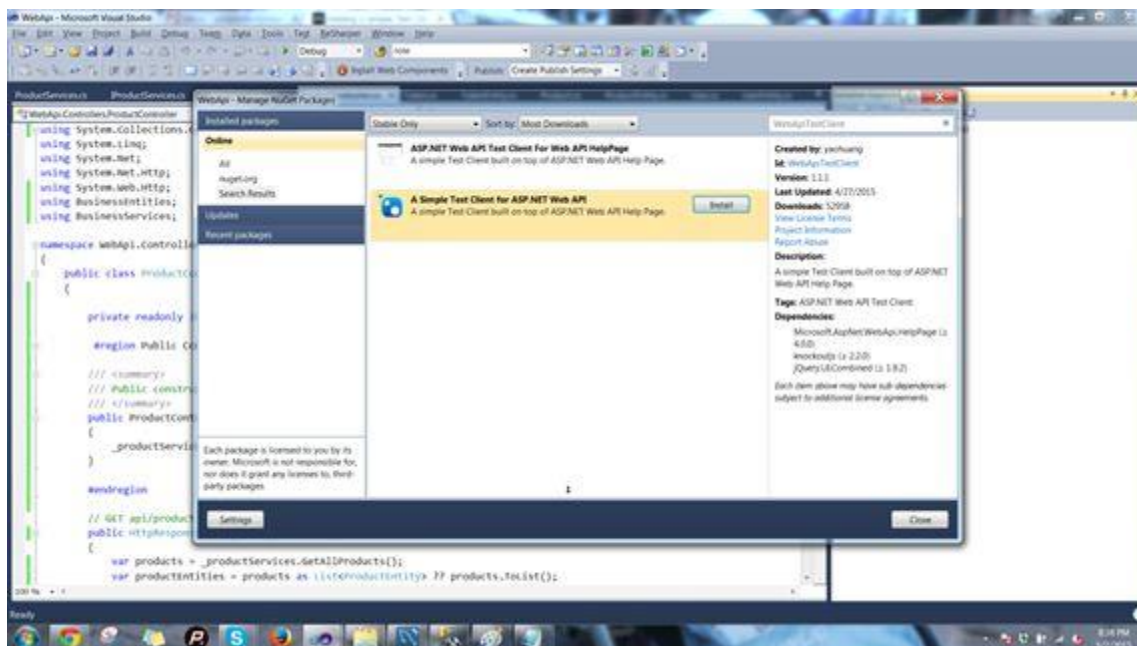
```

Just run the application, we get:

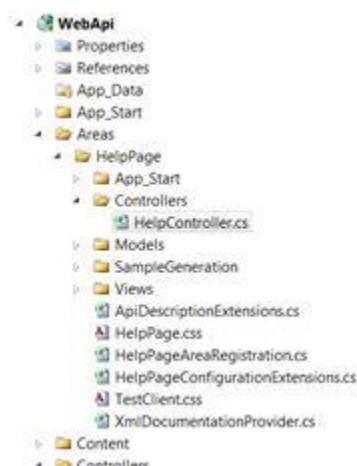


We already have our test client added, but for new readers, I am just again explaining how to add a test client to our API project.

Just go to Manage NuGet Packages by right-clicking the WebAPI project and type WebAPITestClient into the searchbox in online packages as in the following:

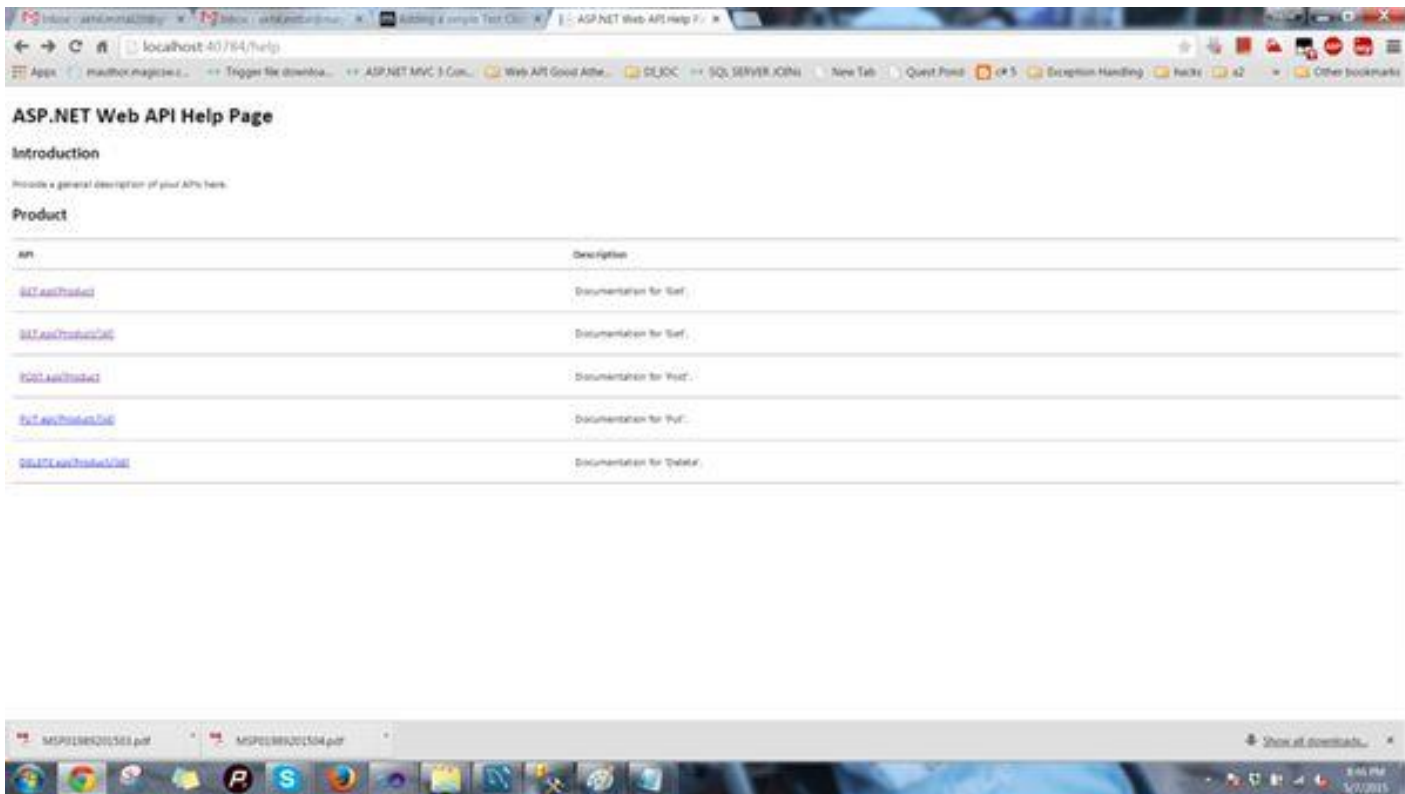


You'll get “A simple Test Client for ASP.NET Web API”, just add it. You'll get a help controller in Areas -> HelpPage like shown below.



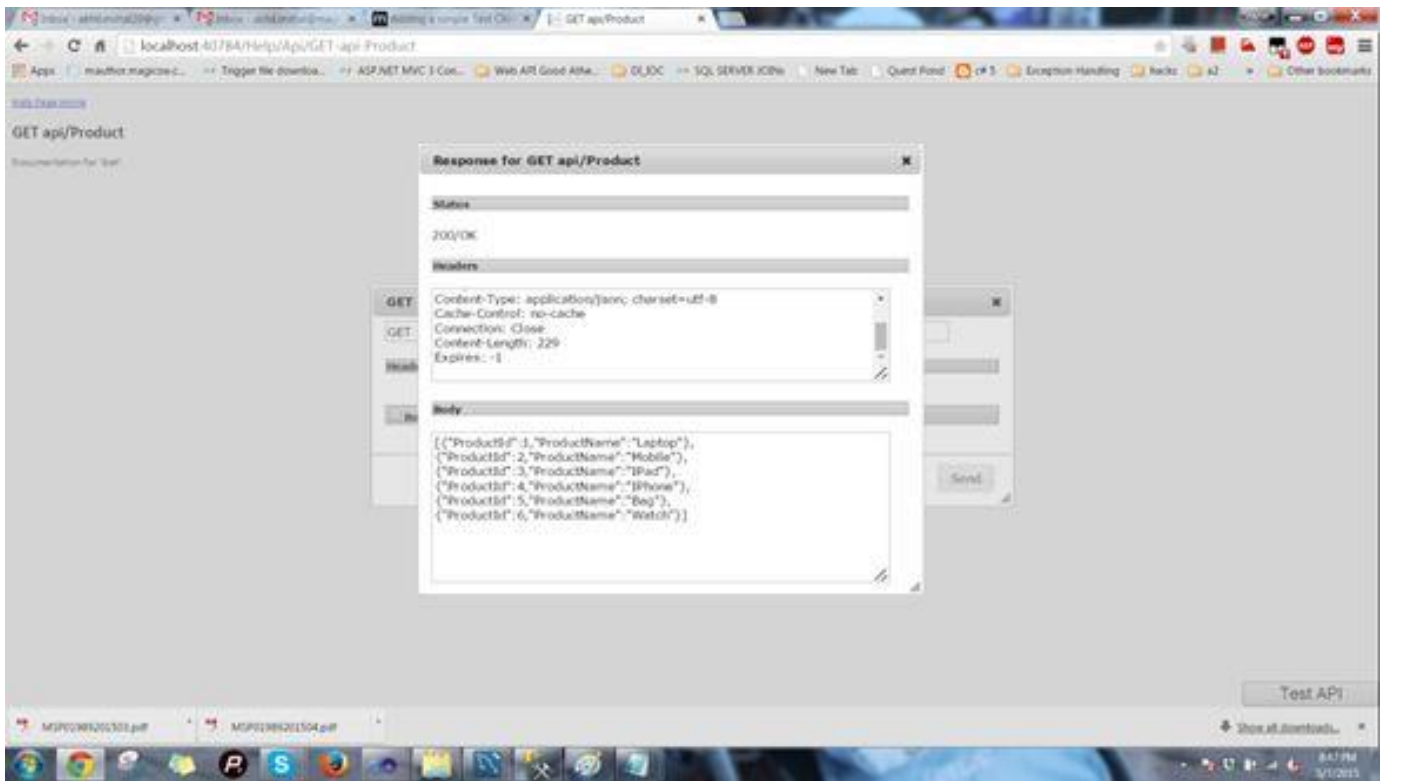
I have already provided the database scripts and data in previous chapter, you can use that.

Append “/help” in the application URL and you'll get the test client as in the following:

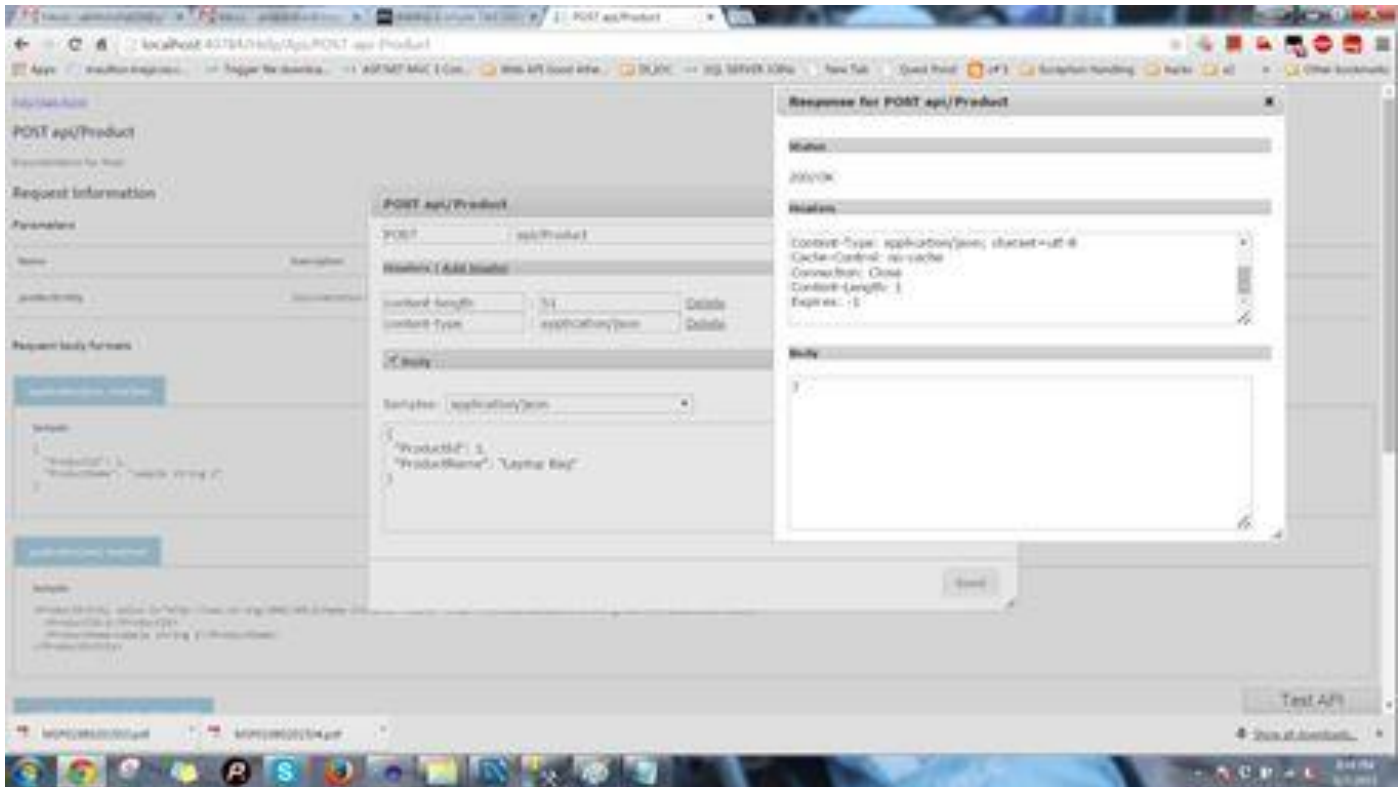


You can test each service by clicking on it.

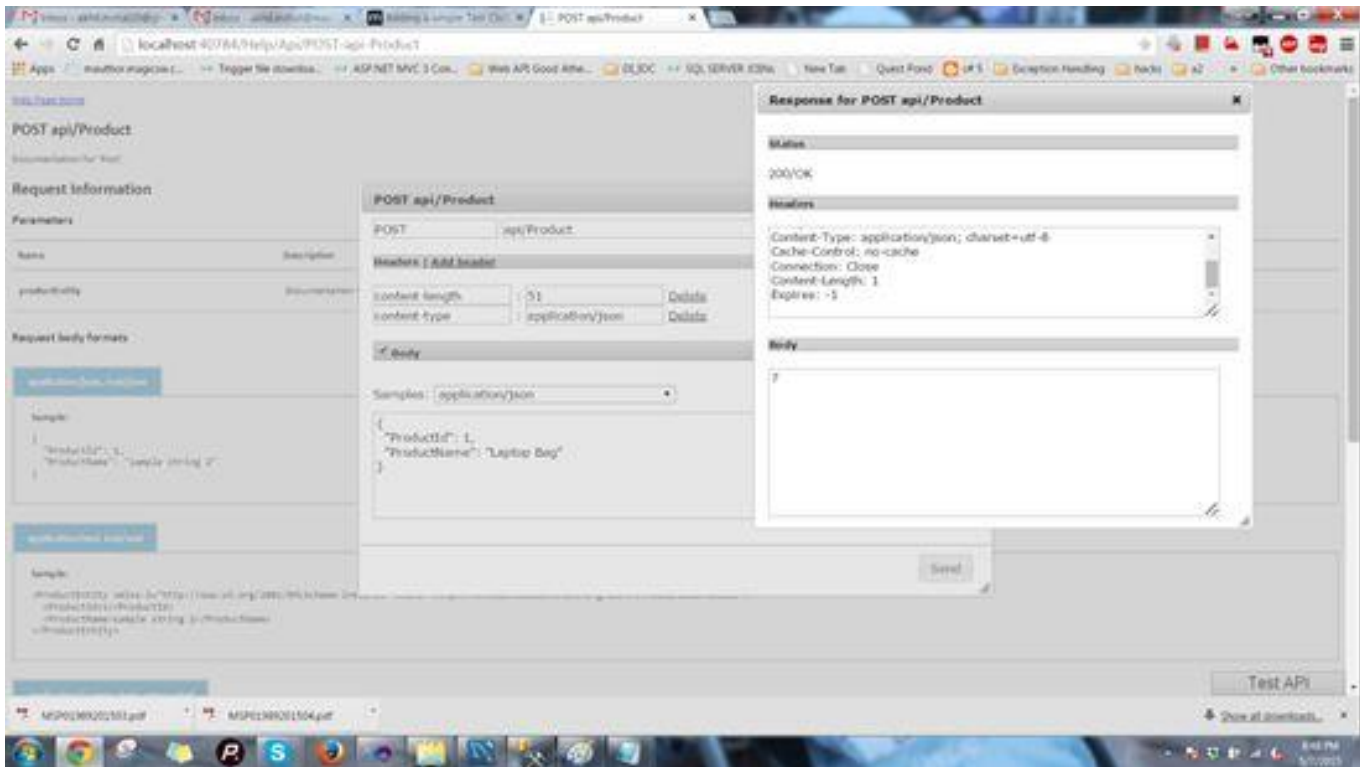
Service for GetAllProduct:



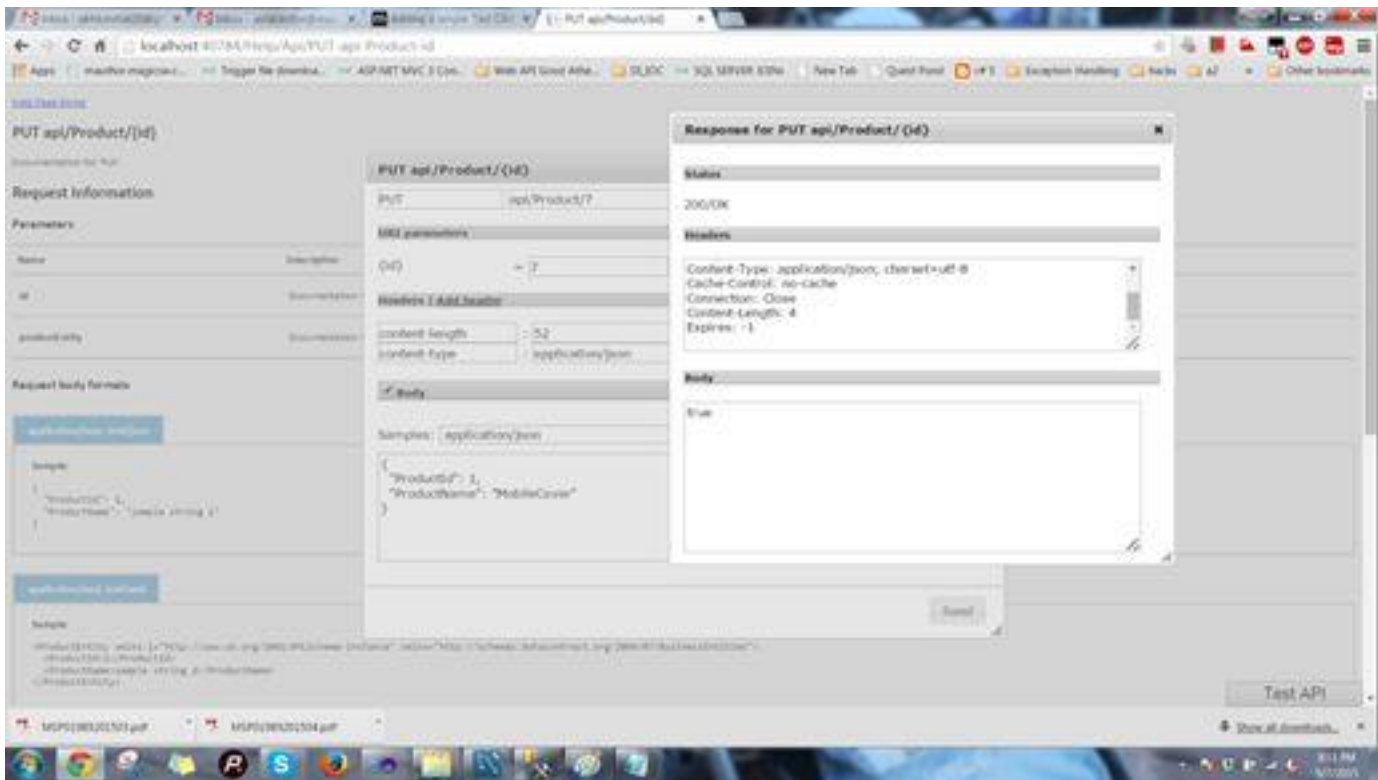
To create a new product:



In the database, we get a new product as in the following:



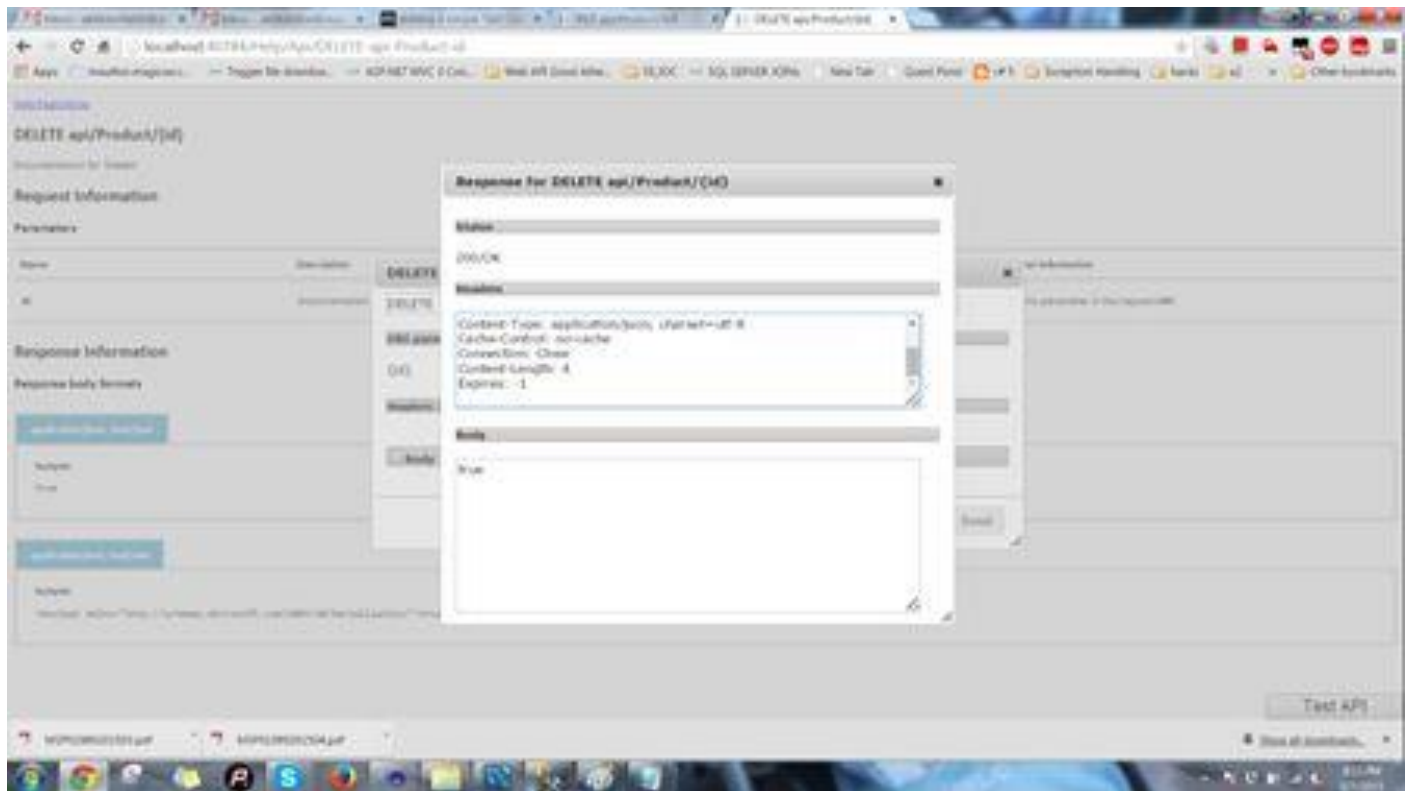
Update product:



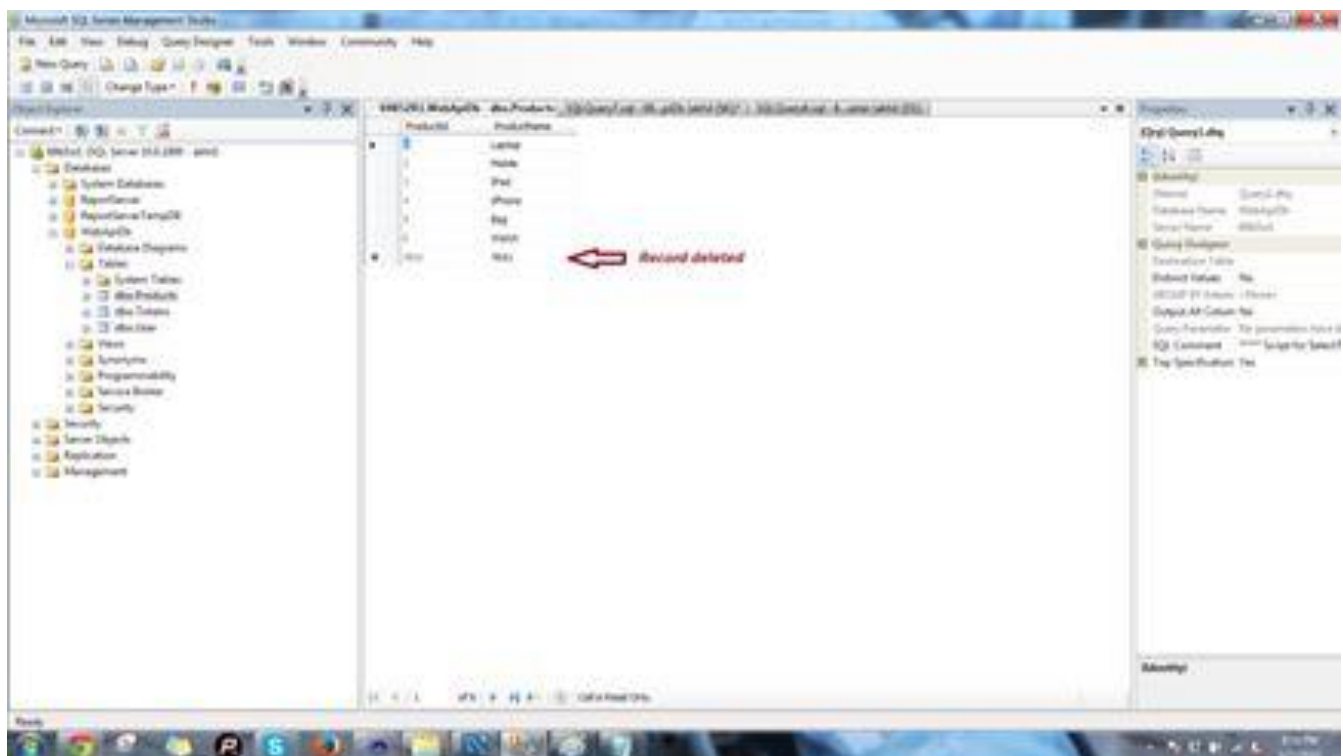
We get the following in the database:

69B5ZR1.WebApiDb - dbo.Products		
ProductID	ProductName	
1	Laptop	
2	Mobile	
3	IPad	
4	IPhone	
5	Bag	
6	Watch	
7	MobileCover	← Updated record
NULL	NULL	

Delete product:



In the database:



Job done.

Design Flaws

What if I say there are still flaws in this design, the design is still not loosely coupled.

Do you remember what we decided when writing our first application?

Our API talks to services and services talk to the DataModel. We'll never allow the DataModel talk to the APIs for security reasons. But did you notice that when we were registering the type in the Bootstrapper class, we also registered the type of UnitOfWork? That means we added the DataModel as a reference to our API project. This is a design problem. We tried to resolve the dependency of a dependency by violating our design and compromising security.

In next chapter, we'll overcome this situation; we'll try to resolve the dependency and its dependency without violating our design and compromising security. In fact, we'll make it more secure and loosely coupled.

Conclusion

We now know how to use Unity container to resolve a dependency and perform Inversion of Control. But still there are some flaws in this design. In next chapter I'll try to make the system stronger.

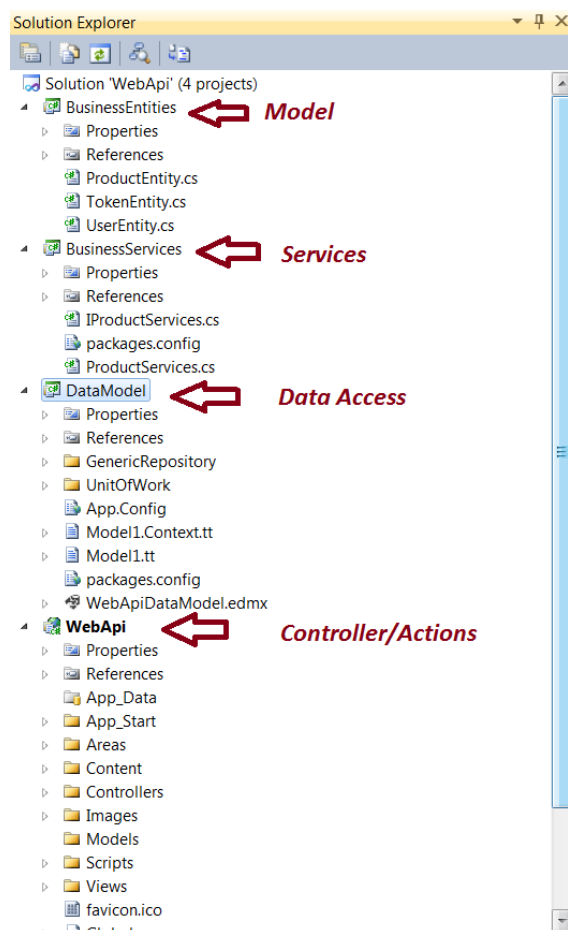
Source Code

You can also download the source code from [GitHub](#). Add the required packages, if they are missing in the source code.

Resolve Dependency of Dependencies Using Inversion of Control & Dependency Injection in ASP.Net Web APIs with Unity Container and Managed Extensibility Framework (MEF)

Existing Design and Problem

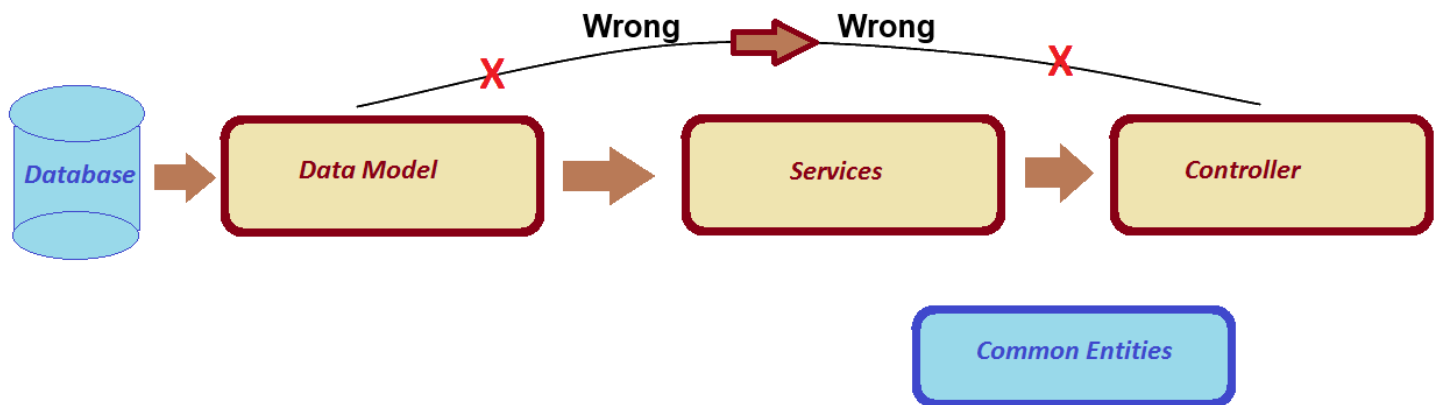
We already have an existing design. If you open the solution, you'll get to see the structure as mentioned below,



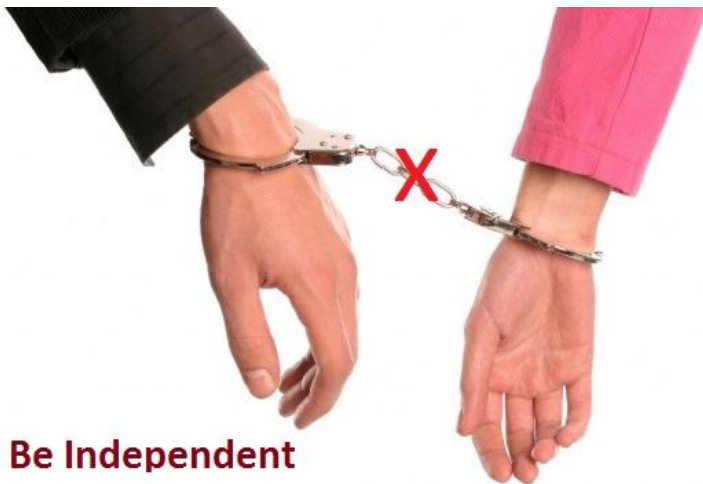
We tried to design a loosely coupled architecture in the following way,

- DataModel (responsible for communication with database) : Only talks to service layer.
- Services (acts as a business logic layer between REST endpoint and data access) : Communicates between REST endpoint and DataModel.
- REST API i.e. Controllers: Only talks to services via the interfaces exposed.

But when we tried to resolve the dependency of UnitOfWork from Services, we had to reference DataModel dll in to our WebAPI project; this violated our system like shown in following image,



In this chapter we'll try to resolve dependency (data model) of a dependency (services) from our existing solution. My controller depended on services and my services depended on data model. Now we'll design an architecture in which components will be independent of each other in terms of object creation and instantiation. To achieve this we'll make use of [MEF\(Managed Extensibility Framework\)](#) along with Unity Container and reflection.



Be Independent

Image source : http://o5.com/wp-content/uploads/2010/08/dreamstime_15583711-550x371.jpg

Ideally, we should not be having the below code in our Bootstrapper class,

```

container.RegisterType<IProductServices,
ProductServices>().RegisterType<UnitOfWork>(new HierarchicalLifetimeManager());
  
```


Managed Extensibility Framework (MEF)

You can have a read about Unity from [msdn](#) link. I am just quoting some lines from msdn,

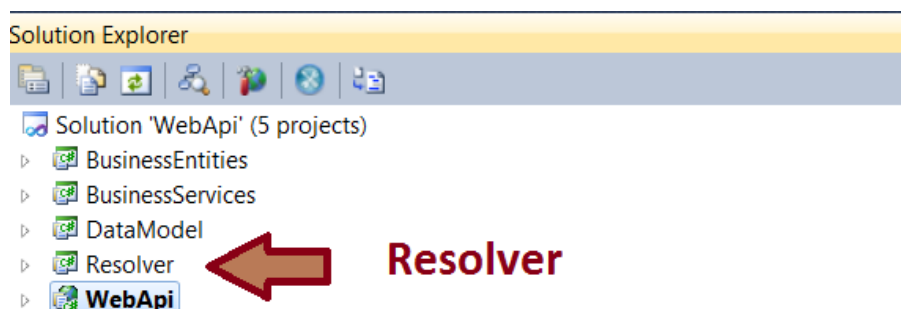
“The Managed Extensibility Framework or MEF is a library for creating lightweight, extensible applications. It allows application developers to discover and use extensions with no configuration required. It also lets extension developers easily encapsulate code and avoid fragile hard dependencies. MEF not only allows extensions to be reused within applications, but across applications as well.”

“MEF is an integral part of the .NET Framework 4, and is available wherever the .NET Framework is used. You can use MEF in your client applications, whether they use Windows Forms, WPF, or any other technology, or in server applications that use ASP.NET.”

Creating a Dependency Resolver with Unity and MEF

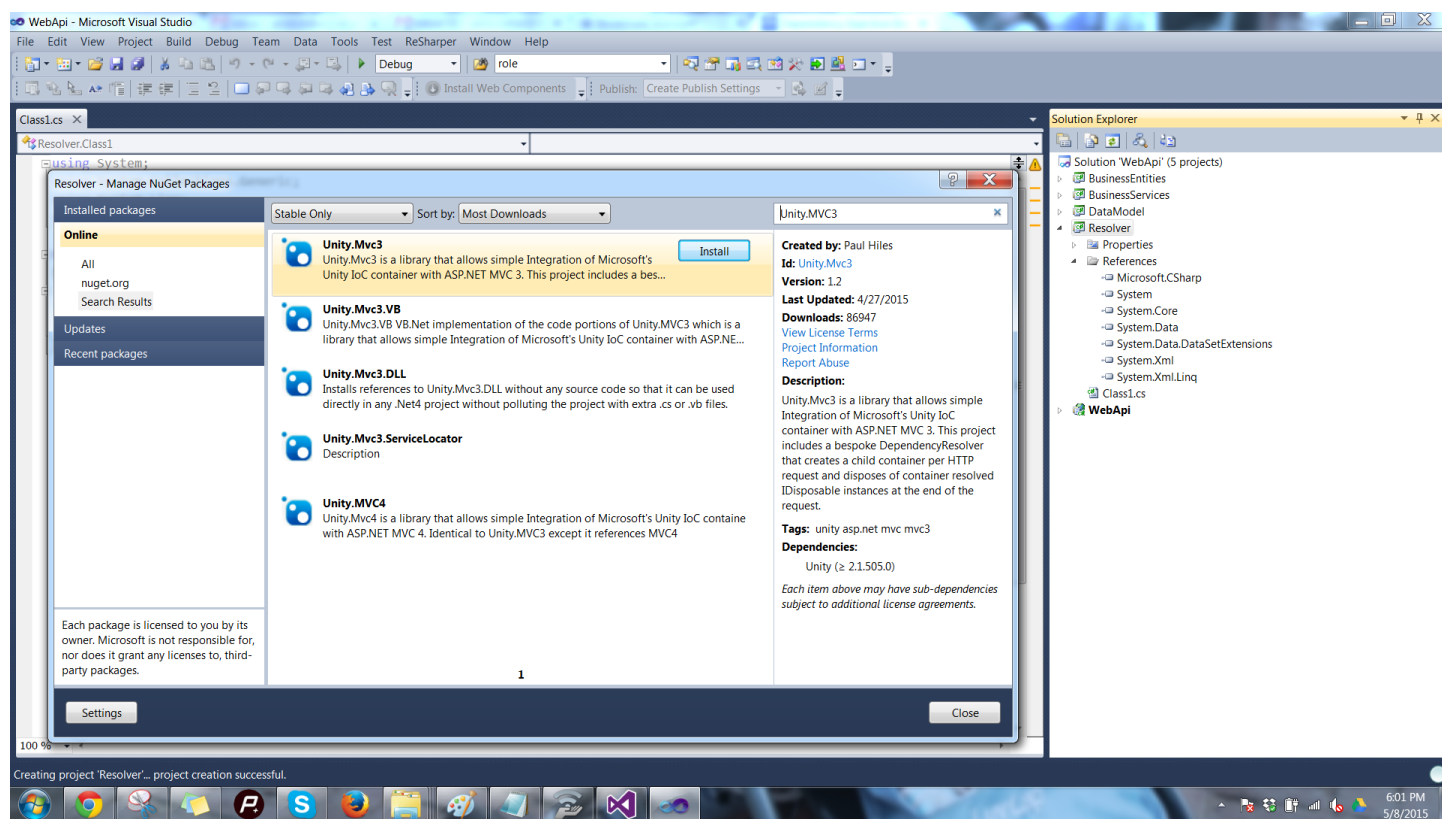
Open your Visual studio, I am using VS 2010, you can use VS version 2010 or above. Load the solution.

Step 1: Right click solution explorer and add a new project named Resolver,



I have intentionally chosen this name, and you already know it why 😊

Step 2: Right click Resolver project and click on ManageNuGetPackage, in the interface of adding new package, search Unity.MVC3 in online library,



Install the package to your solution.

Step 3: Right click resolver project and add a reference to System.ComponentModel.Composition.

You can find the dll into your GAC. I am using framework 4.0, so referring to the same version dll.

ReachFramework	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
sysglobl	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
System.Activities.Core.Presentation	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
System.Activities	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
System.Activities.DurableInstancing	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
System.Activities.Presentation	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
System.AddIn.Contract	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
System.AddIn	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
System.ComponentModel.Composition	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
System.ComponentModel.DataAnnotations	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
System.Configuration	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...
System.Configuration.Install	4.0.0.0	v4.0.30319	C:\Program Files (x86)\Ref...

This dll is the part of MEF and is already installed with .net Framework 4.0 in the system GAC. This dll provides classes that are very core of MEF.

Step 4: Just add an interface named IComponent to Resolver project that contains the initialization method named Setup. We'll try to implement this interface into our Resolver class that we'll create in our other projects like DataModel, Services and WebApi.

`namespace` Resolver

```
{
    /// <summary>
    /// Register underlying types with unity.
    /// </summary>

    public interface IComponent
    {
    }
}
```


Step 5: Before we declare our Setup method, just add one more interface responsible for serving as a contract to register types. I name this interface as IRegisterComponent,

```
namespace Resolver
{
    /// <summary>
    /// Responsible for registering types in unity configuration by implementing IComponent
    /// </summary>
    public interface IRegisterComponent
    {
        /// <summary>
        /// Register type method
        /// </summary>
        /// <typeparam name="TFrom"></typeparam>
        /// <typeparam name="TTo"></typeparam>
        /// <param name="withInterception"></param>
        void RegisterType<TFrom, TTo>(bool withInterception = false) where TTo : TFrom;

        /// <summary>
        /// Register type with container controlled life time manager
        /// </summary>
    }
}
```



```

    /// <typeparam name="TFrom"></typeparam>

    /// <typeparam name="TTo"></typeparam>

    /// <param name="withInterception"></param>

    void RegisterTypeWithControlledLifeTime<TFrom, TTo>(bool withInterception = false)
    where TTo : TFrom;

}

```

In this interface I have declared two methods, one RegisterType and other in to RegisterType with Controlled life time of the object, i.e. the life time of an object will be hierarchal in manner. This is kind of same like we do in Unity.

Step 6: Now declare Setup method on our previously created IComponent interface, that takes instance of IRegisterComponent as a parameter,

```
void SetUp(IRegisterComponent registerComponent);
```

So our IComponent interface becomes,

```

namespace Resolver

{

    /// <summary>

    /// Register underlying types with unity.

    /// </summary>

```



```
public interface IComponent
{
    void SetUp(IRegisterComponent registerComponent);
}
}
```

Step 6: Now we'll write a packager or you can say a wrapper over MEF and Unity to register types/ components. This is the core MEF implementation. Create a class named ComponentLoader, and add following code to it,

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using Microsoft.Practices.Unity;

using System.ComponentModel.Composition.Hosting;

using System.ComponentModel.Composition.Primitives;

using System.Reflection;

namespace Resolver
{
    public static class ComponentLoader
    {

```



```

public static void LoadContainer(IUnityContainer container, string path, string pattern)
{
    var dirCat = new DirectoryCatalog(path, pattern);

    var importDef = BuildImportDefinition();

    try
    {
        using (var aggregateCatalog = new AggregateCatalog())
        {
            aggregateCatalog.Catalogs.Add(dirCat);

            using (var compositionContainer = new
CompositionContainer(aggregateCatalog))
            {
                IEnumerable<Export> exports =
compositionContainer.GetExports(importDef);

                IEnumerable<IComponent> modules =
                    exports.Select(export => export.Value as IComponent).Where(m => m !=
null);

                var registerComponent = new RegisterComponent(container);

                foreach (IComponent module in modules)
                {
                    module.SetUp(registerComponent);
                }
            }
        }
    }
}

```



```

        }

    }

}

catch (ReflectionTypeLoadException typeLoadException)
{
    var builder = new StringBuilder();

    foreach (Exception loaderException in typeLoadException.LoaderExceptions)
    {
        builder.AppendFormat("{0}\n", loaderException.Message);
    }

    throw new TypeLoadException(builder.ToString(), typeLoadException);
}

}

private static ImportDefinition BuildImportDefinition()
{
    return new ImportDefinition(
        def => true, typeof(IComponent).FullName, ImportCardinality.ZeroOrMore, false,
false);
}

}

```



```
internal class RegisterComponent : IRegisterComponent
{
    private readonly IUnityContainer _container;

    public RegisterComponent(IUnityContainer container)
    {
        this._container = container;

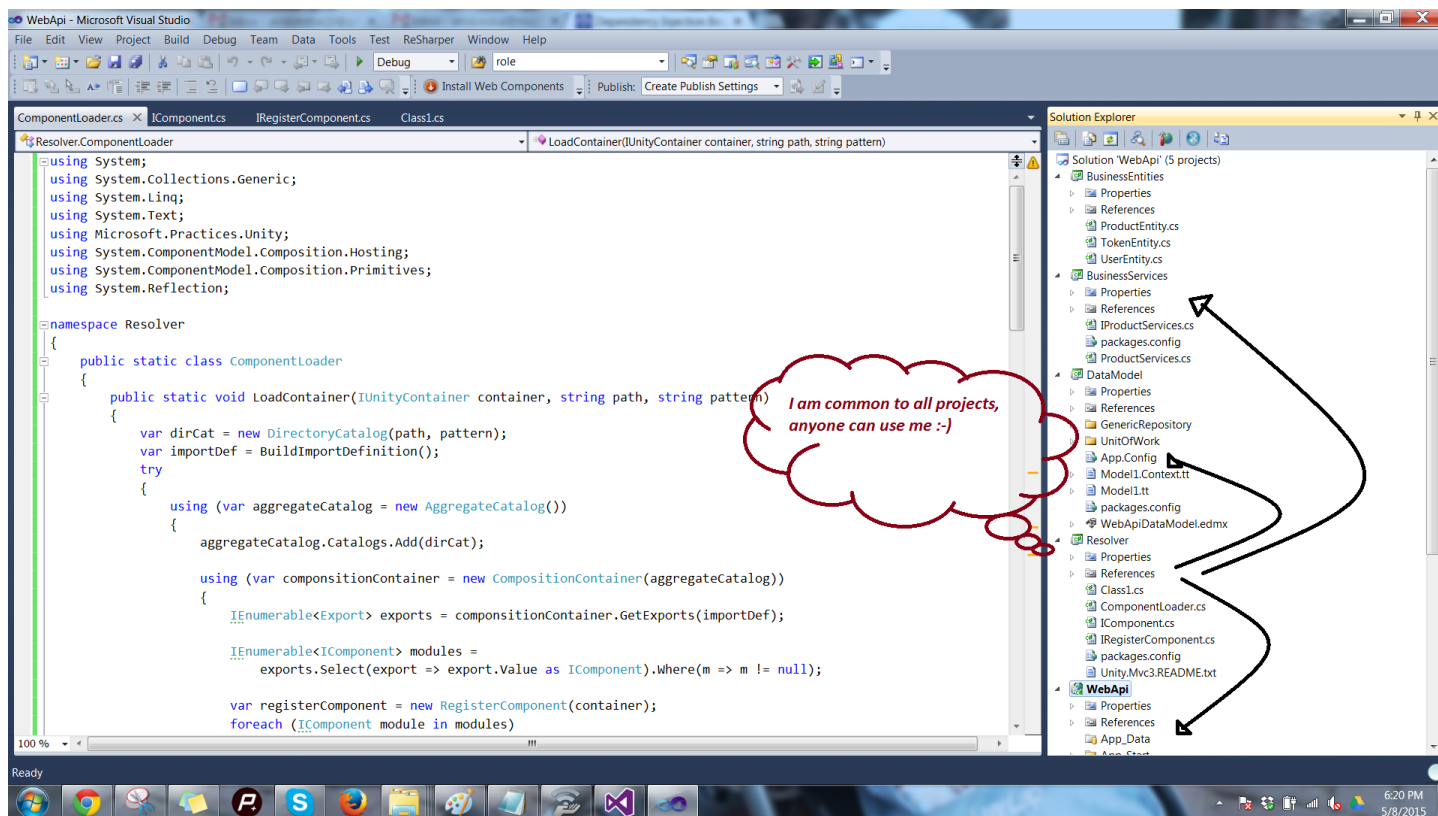
        //Register interception behaviour if any
    }

    public void RegisterType<TFrom, TTo>(bool withInterception = false) where TTo :
    TFrom
    {
        if (withInterception)
        {
            //register with interception
        }
        else
        {
            this._container.RegisterType<TFrom, TTo>();
        }
    }
}
```



```
public void RegisterTypeWithControlledLifeTime<TFrom, TTo>(bool withInterception =  
false) where TTo : TFrom  
  
{  
  
    this._container.RegisterType<TFrom, TTo>(new  
ContainerControlledLifetimeManager());  
  
}  
  
}  
  
}
```

Step 7 :Now our Resolver wrapper is ready.Build the project and add its reference to DataModel, BusinessServices and WebApi project like shown below,



Setup Business Services

We already have added reference of Resolver in BusinessServices project. We agreed to implement IComponent interface in each of our project.

So create a class named DependencyResolver and implement IComponent interface into it, we make use of reflection too to import IComponent type. So add a class and add following code to that DependencyResolver class,

```
using System.ComponentModel.Composition;
```

```
using DataModel;
```

```
using DataModel.UnitOfWork;
```

```
using Resolver;
```

```
namespace BusinessServices
```



```

{

[Export(typeof(IComponent))]

public class DependencyResolver : IComponent

{

    public void SetUp(IRegisterComponent registerComponent)

    {

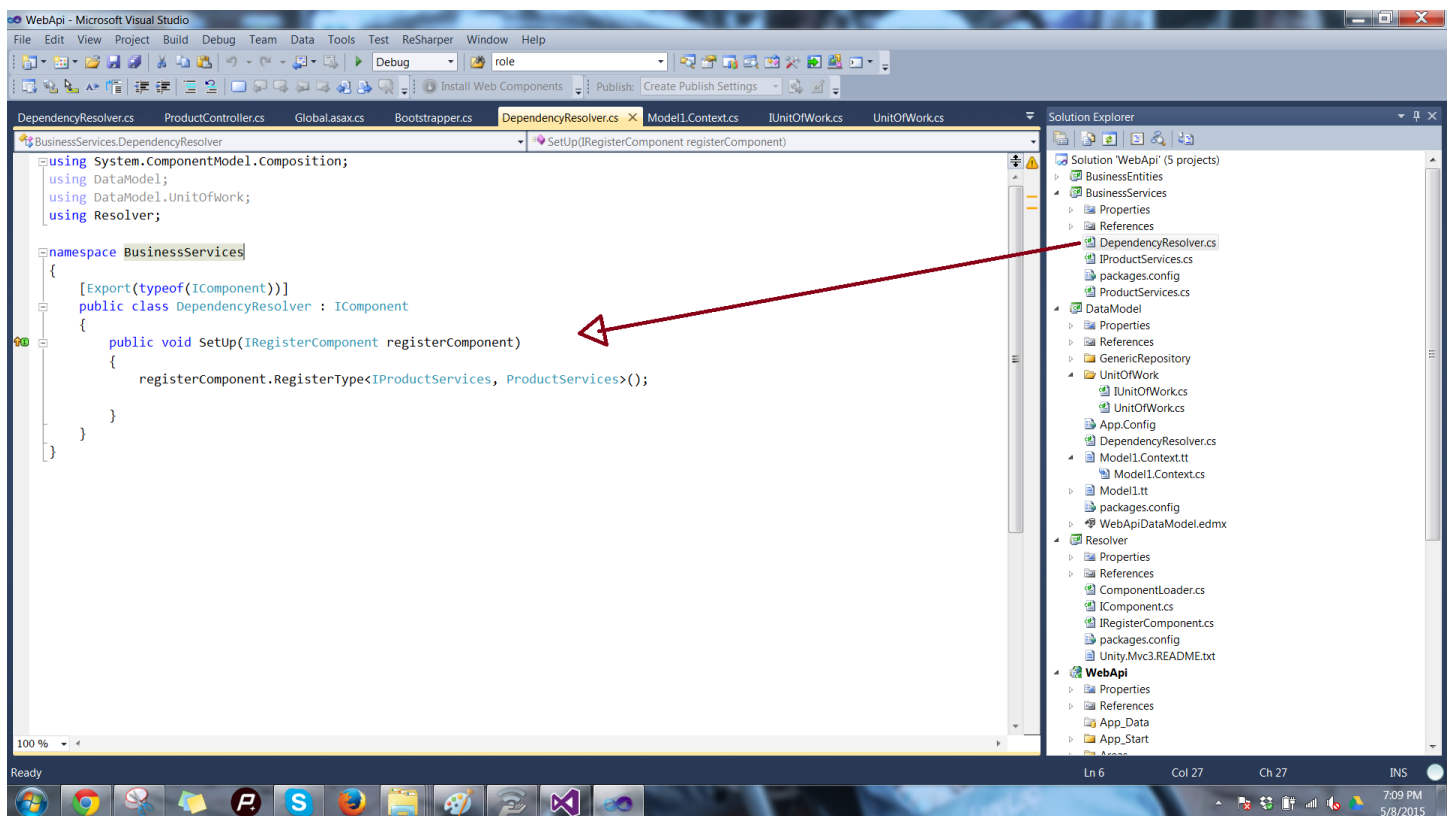
        registerComponent.RegisterType<IProductServices, ProductServices>();

    }

}

}

```



Note that we have implemented Setup method and in the same method we registered type for my ProductService.

All of the existing code base remains same. We don't need to touch the IProductServices interface or ProductServices class.

Setup DataModel

We have added Resolver project reference to DataModel project as well. So we'll try to register the type of UnitOfWork in this project. We proceed in same fashion, just add a DependencyResolver class and implement its Setup method to register type of UnitOfWork. To make the code more readable and standard, I made a change. I just added an interface for UnitOfWork and named it IUnitOfWork. Now my UnitOfWork class derives from this, you can do this exercise in earlier versions of projects we discussed in first two articles.

So my IUnitOfWork contains declaration of a single public method in UnitOfWork,

`namespace DataModel.UnitOfWork`

```
{  
  
    public interface IUnitOfWork  
  
    {  
  
        /// <summary>  
        /// Save method.  
        /// </summary>  
  
        void Save();  
  
    }  
  
}
```


Now register the type for UnitOfWork in DependencyResolver class, our class becomes as shown below,

```
using System.ComponentModel.Composition;
```

```
using System.Data.Entity;
```

```
using DataModel.UnitOfWork;
```

```
using Resolver;
```

```
namespace DataModel
```

```
{
```

```
    [Export(typeof(IComponent))]
```

```
    public class DependencyResolver : IComponent
```

```
    {
```

```
        public void SetUp(IRegisterComponent registerComponent)
```

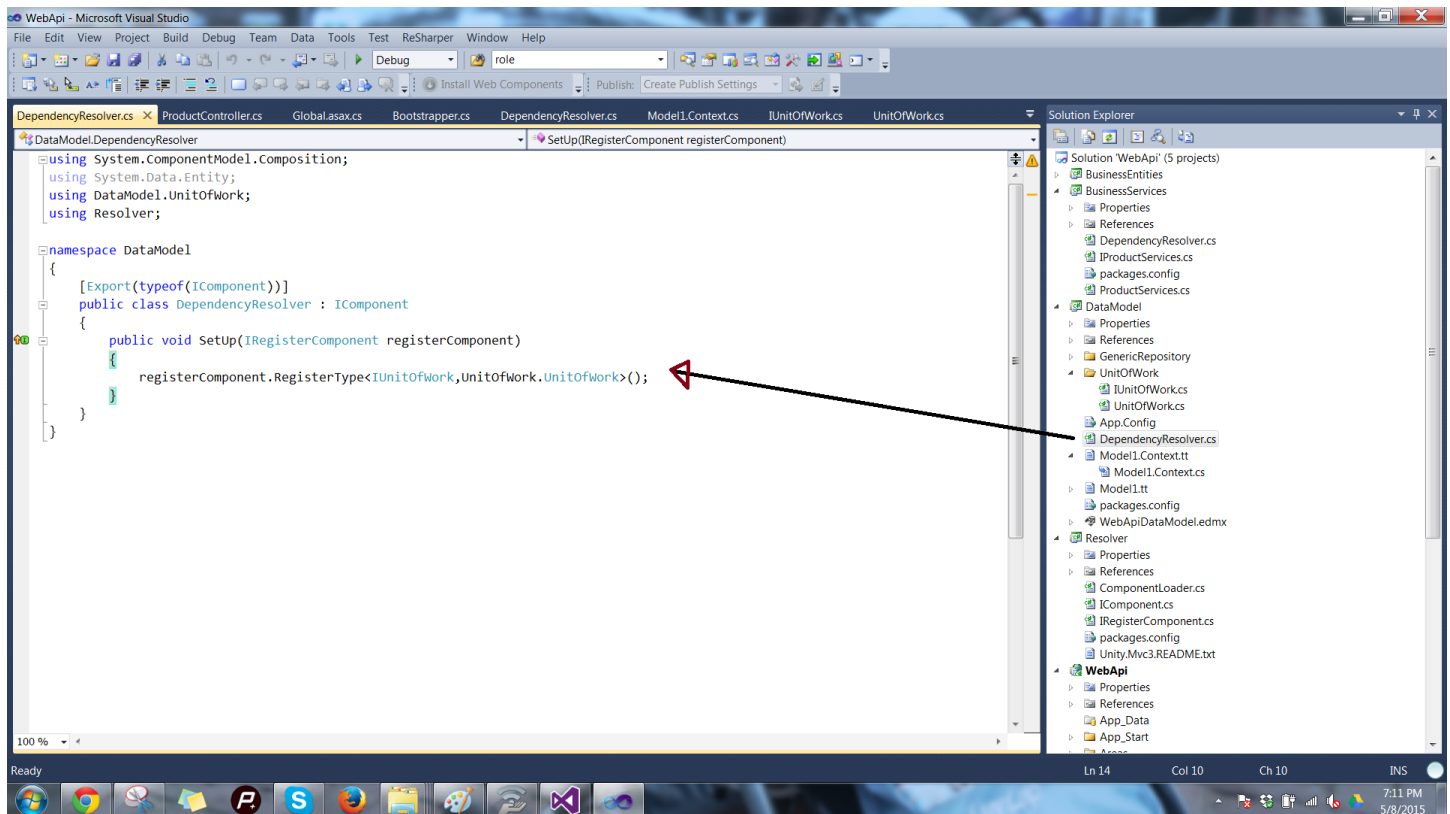
```
        {
```

```
            registerComponent.RegisterType<IUnitOfWork, UnitOfWork.UnitOfWork>();
```

```
        }
```

```
    }
```

```
}
```

Again, no need to touch any existing code of this project.

Setup REST endpoint / WebAPI project

Our 90% of the job is done.



We now need to setup or WebAPI project. We'll not add any DependencyResolver class in this project. We'll invert the calling mechanism of layers in Bootstrapper class that we already have, so when you open your bootstrapper class, you'll get the code something like,

```
using System.Web.Http;
```

```
using System.Web.Mvc;
```



```
using BusinessServices;
```

```
using DataModel.UnitOfWork;
```

```
using Microsoft.Practices.Unity;
```

```
using Unity.Mvc3;
```

```
namespace WebApi
```

```
{
```

```
    public static class Bootstrapper
```

```
    {
```

```
        public static void Initialise()
```

```
        {
```

```
            var container = BuildUnityContainer();
```

```
            DependencyResolver.SetResolver(new UnityDependencyResolver(container));
```

```
            // register dependency resolver for WebAPI RC
```

```
            GlobalConfiguration.Configuration.DependencyResolver = new  
Unity.WebApi.UnityDependencyResolver(container);
```

```
        }
```

```
        private static IUnityContainer BuildUnityContainer()
```

```
        {
```

```
            var container = new UnityContainer();
```



```
// register all your components with the container here

// it is NOT necessary to register your controllers

// e.g. container.RegisterType<ITestService, TestService>();

container.RegisterType<IProductServices,
ProductServices>().RegisterType<UnitOfWork>(new HierarchicalLifetimeManager());

return container;

}

}

}
```

Now, we need to change the code base a bit to make our system loosely coupled. Just remove the reference of DataModel from WebAPI project.

We don't want our DataModel to be exposed to WebAPI project, that was our aim though, so we cut down the dependency of DataModel project now.



Add following code of Bootstrapper class to the existing Bootstarpper class,

```
using System.Web.Http;

//using DataModel.UnitOfWork;

using Microsoft.Practices.Unity;

using Resolver;

using Unity.Mvc3;

namespace WebApi
{
    public static class Bootstrapper
    {
        public static void Initialise()
        {
            var container = BuildUnityContainer();

            System.Web.Mvc.DependencyResolver.SetResolver(new
            UnityDependencyResolver(container));

            // register dependency resolver for WebAPI RC

            GlobalConfiguration.Configuration.DependencyResolver = new
            Unity.WebApi.UnityDependencyResolver(container);
        }
    }
}
```



```
private static IUnityContainer BuildUnityContainer()
{
    var container = new UnityContainer();

    // register all your components with the container here

    // it is NOT necessary to register your controllers

    // e.g. container.RegisterType<ITestService, TestService>();

    // container.RegisterType<IProductServices,
ProductServices>().RegisterType<UnitOfWork>(new HierarchicalLifetimeManager());

    RegisterTypes(container);

    return container;
}

public static void RegisterTypes(IUnityContainer container)
{
    //Component initialization via MEF

    ComponentLoader.LoadContainer(container, ".\\bin", "WebApi.dll");

    ComponentLoader.LoadContainer(container, ".\\bin", "BusinessServices.dll");
}
```



```

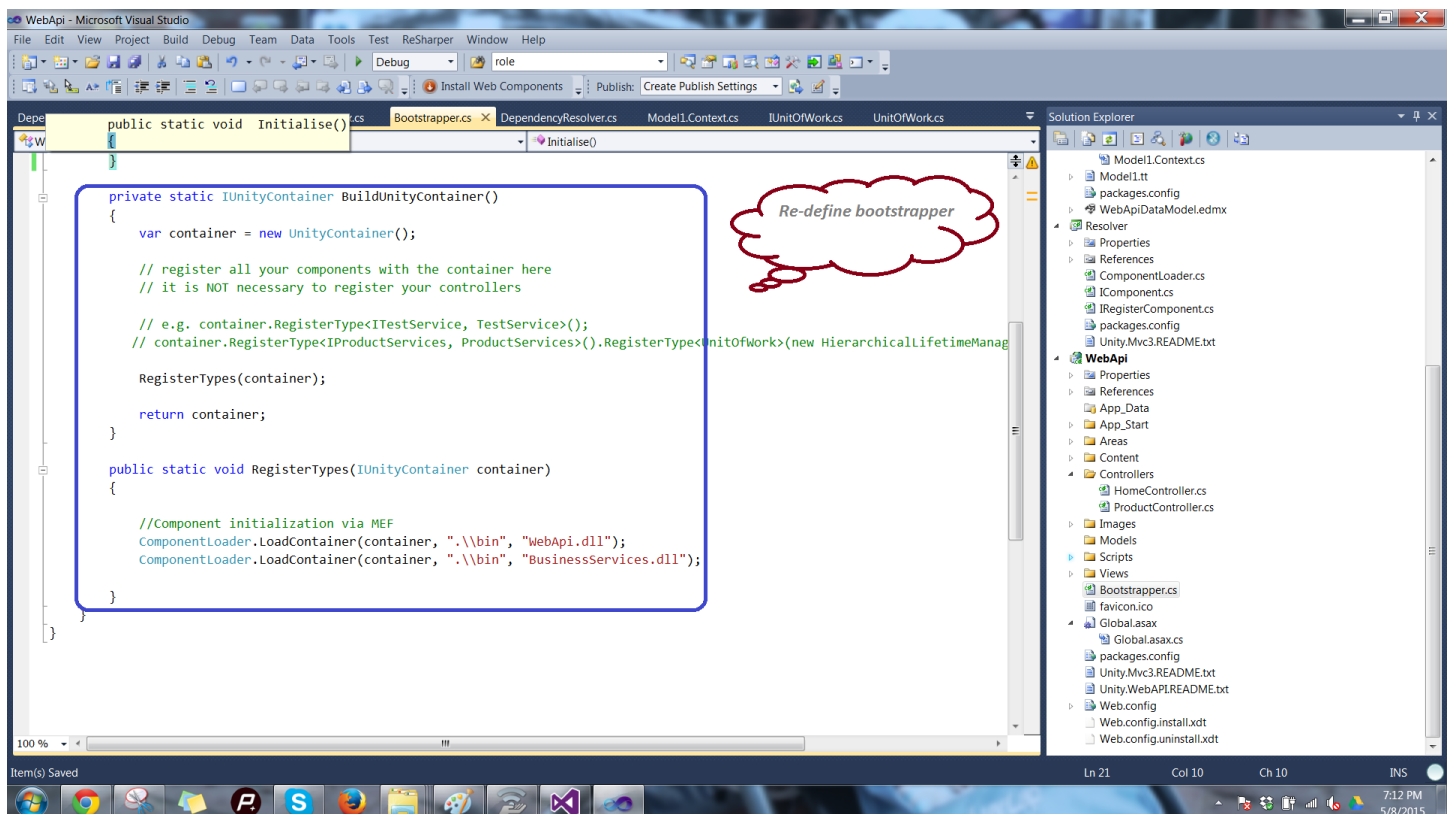
    }

}

}

```

It is kind of redefining Bootstrapper class without touching our existing controller methods. We now don't even have to register type for ProductServices as well, we already did this in BusinessServices project.



Note that in RegisterTypes method we load components/dlls through reflection making use of ComponentLoader. We wrote two lines, first to load WebAPI.dll and another one to load Business Services.dll.

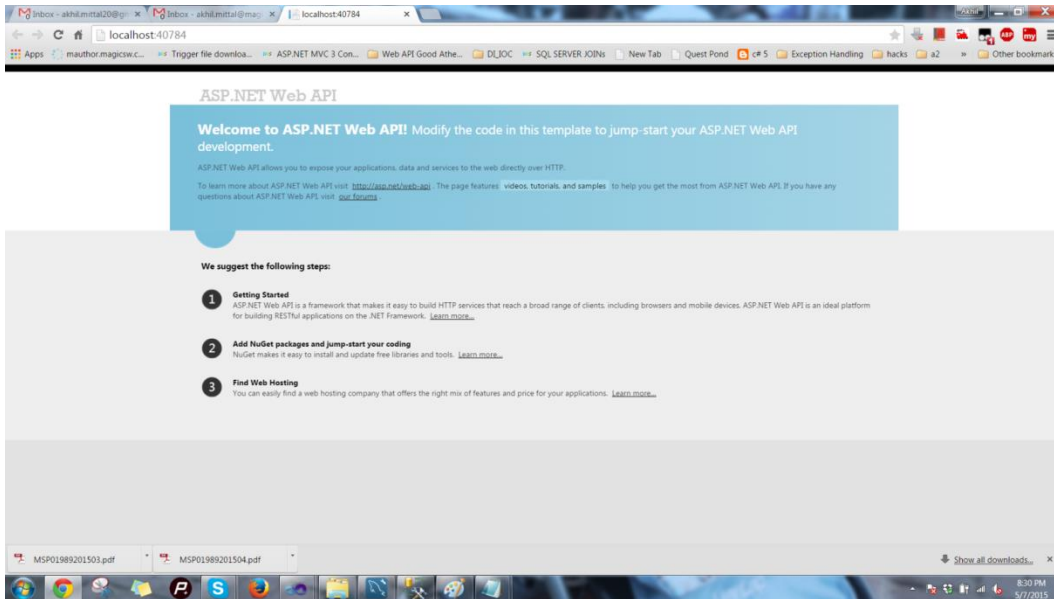
Had the name of BusinessServices.dll be WebAPI.Services.dll, then we would have only written one line of code to load both the WebAPI and BusinessService dll like shown below,

```
ComponentLoader.LoadContainer(container, ".\\bin", "WebApi*.dll");
```

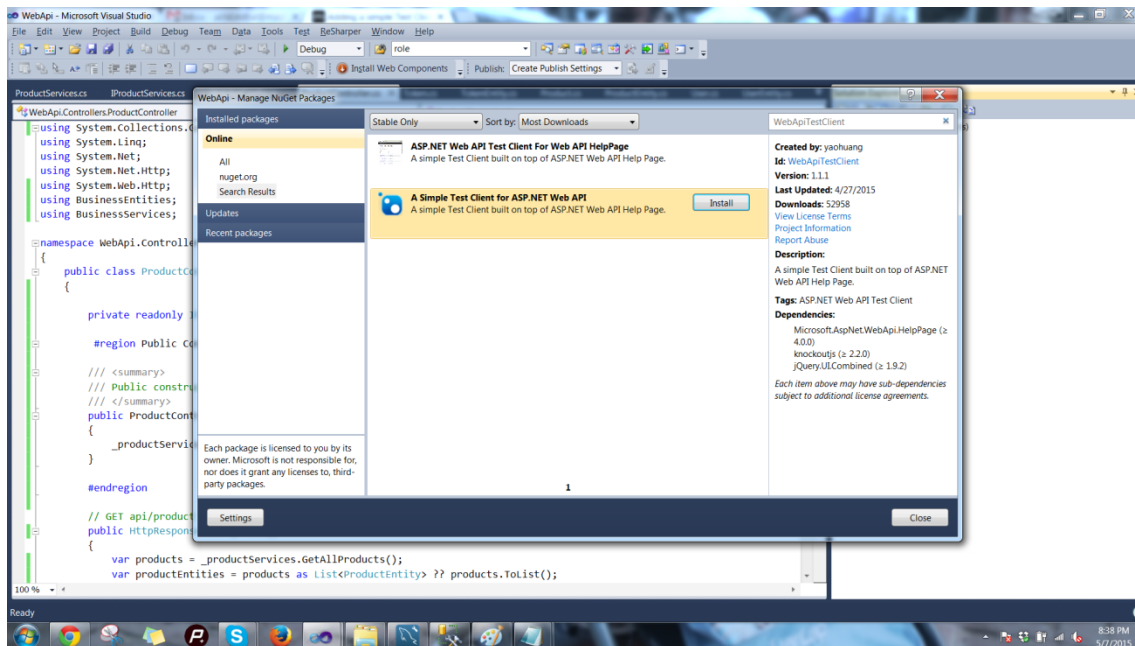

Yes we can make use of Regex.

Running the application

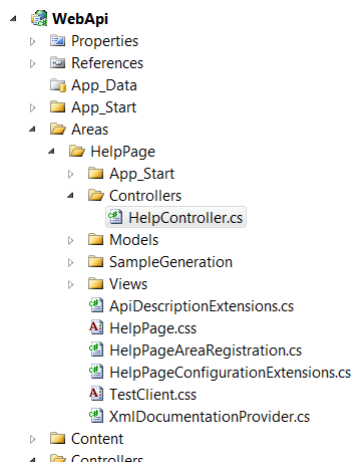
Just run the application, we get,



We already have our test client added, but for new readers, just go to Manage Nuget Packages, by right clicking WebAPI project and type WebAPITestClient in searchbox in online packages,

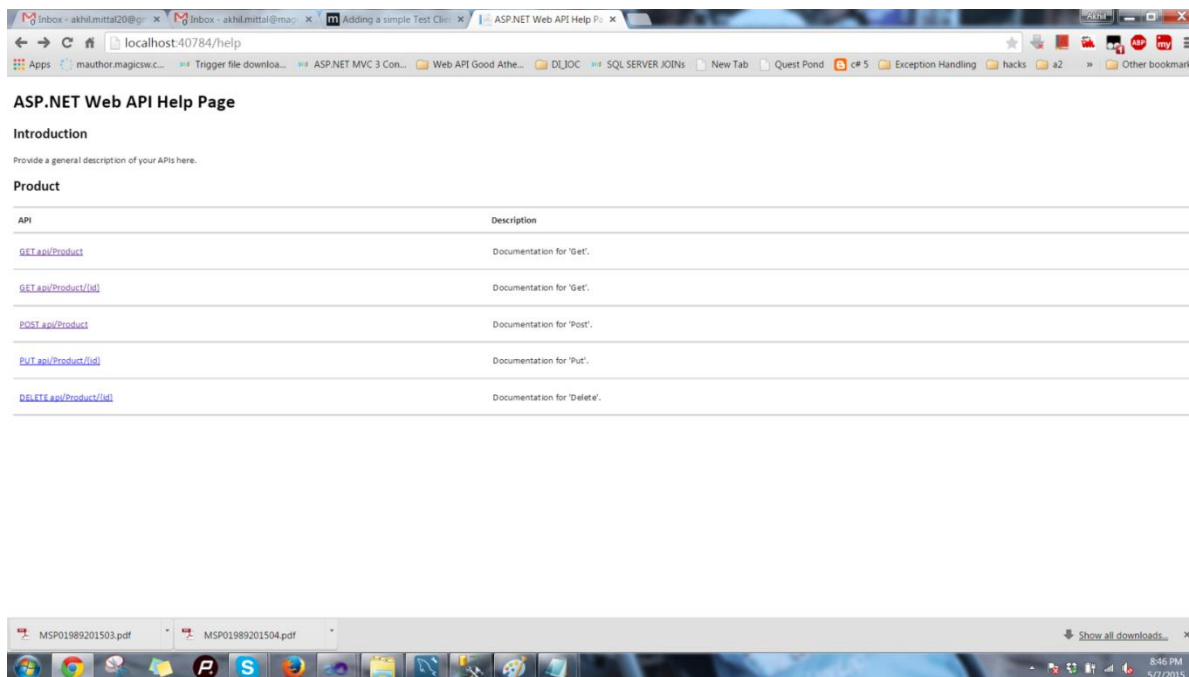


You'll get "A simple Test Client for ASP.NET Web API", just add it. You'll get a help controller in Areas-> HelpPage like shown below,



I have already provided the database scripts and data in my previous article, you can use the same.

Append "/help" in the application url, and you'll get the test client,

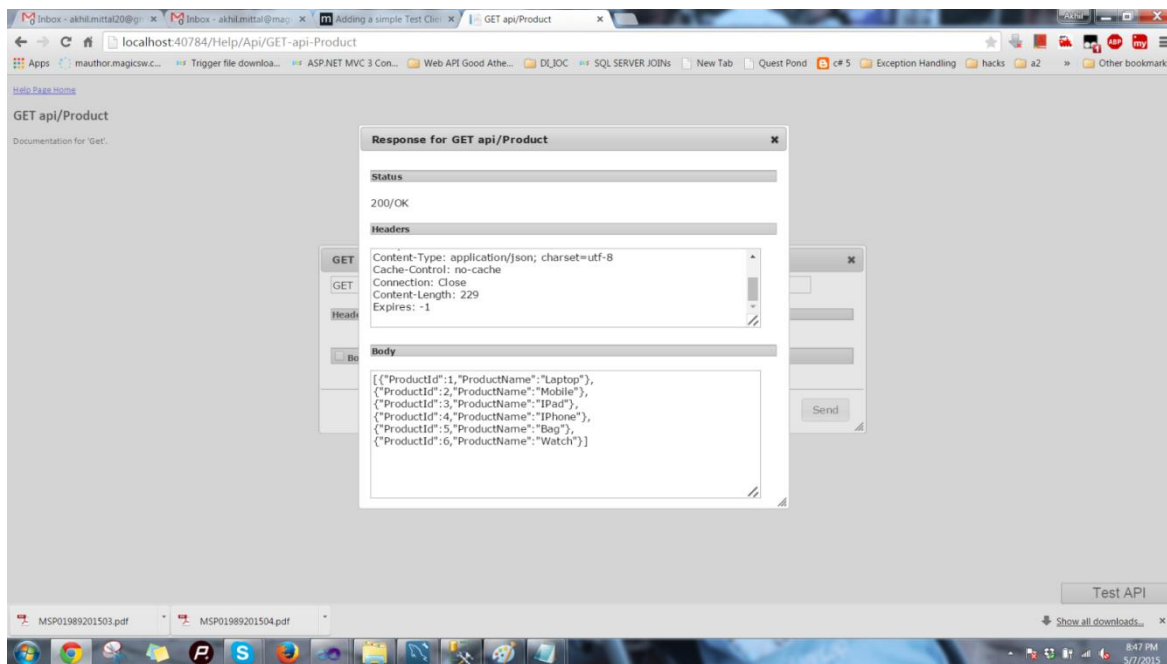


You can test each service by clicking on it. Once you click on the service link, you'll be redirected to test the service page of that particular service. On that page there is a button Test API in the right bottom corner, just press that button to test your service,

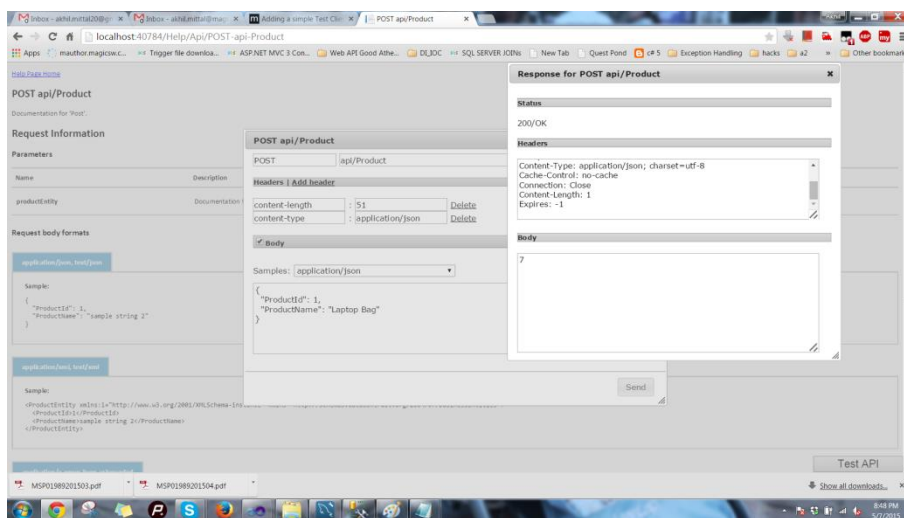
**Press this button to
test the API**



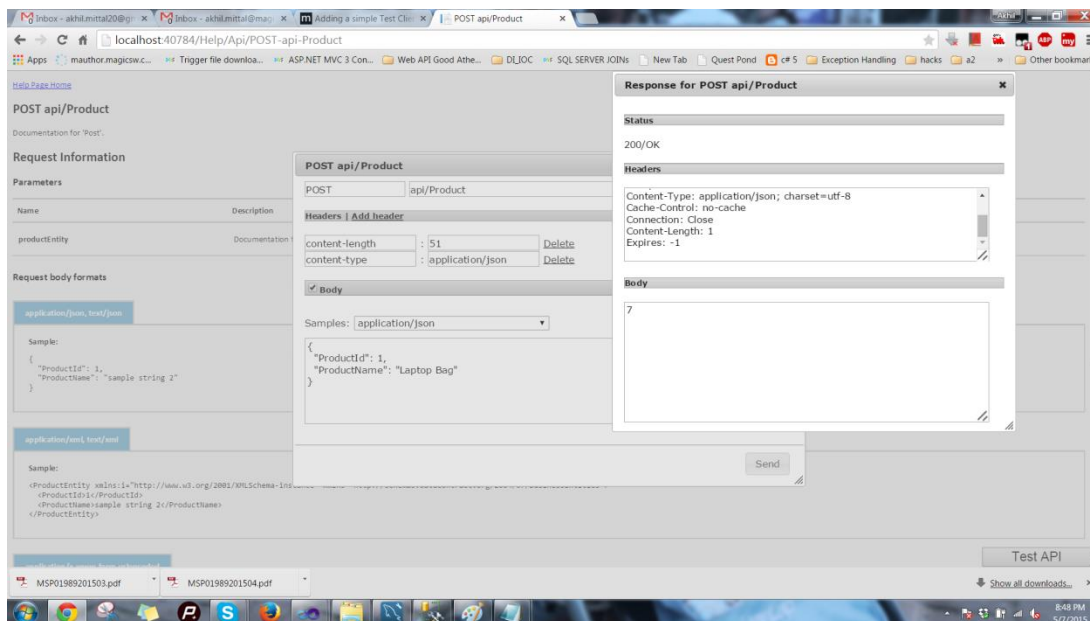
Service for GetAllProduct,



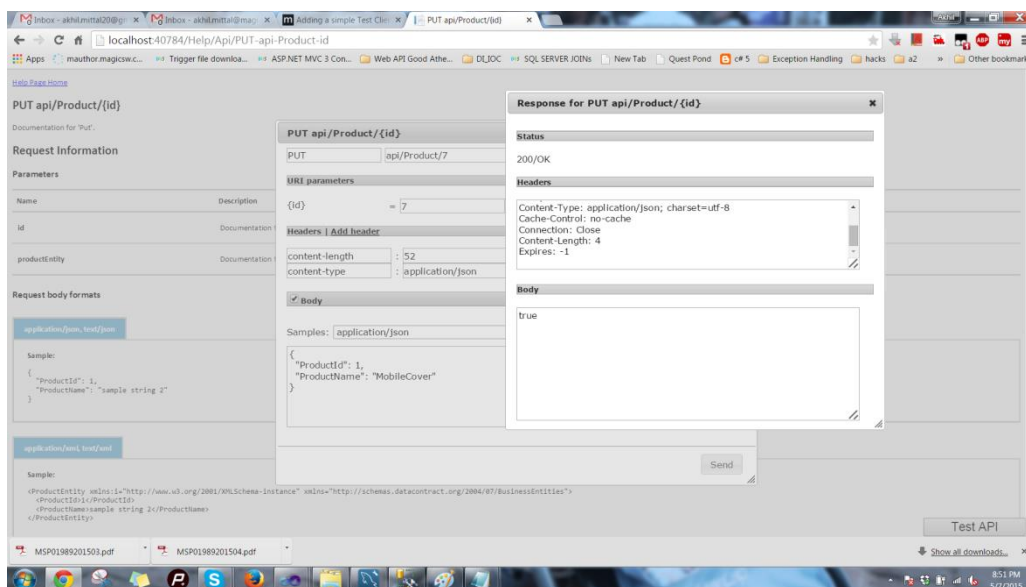
For Create a new product,



In database, we get new product,



Update product:



We get in database,

69B5ZR1.WebApiDb - dbo.Products SQLQuery7.sql - 69...piDb (akhil (56))* SQLC

ProductId	ProductName
1	Laptop
2	Mobile
3	IPad
4	IPhone
5	Bag
6	Watch
7	MobileCover
*	NULL

← Updated record

Delete product:

DELETE api/Product/{id}

Documentation for "Delete".

Request Information

Parameters

Name	Description
id	Documentation

Response Information

Response body formats

application/json; charset=utf-8

Sample: true

application/json; charset=utf-8

Sample: {"boolean":true,"http://schemas.microsoft.com/2003/05/Serialization/"}true

Test API

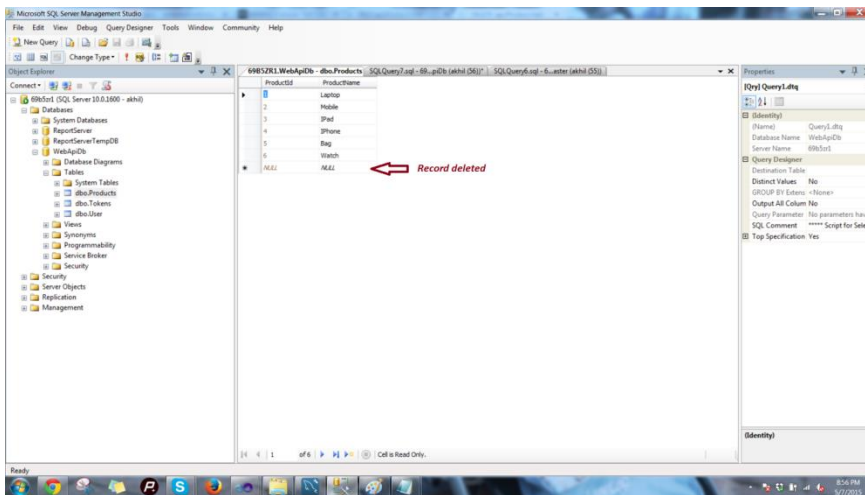
Response for DELETE api/Product/{id}

Status: 200/OK

Headers: Content-Type: application/json; charset=utf-8, Cache-Control: no-cache, Connection: close, Content-Length: 4, Expires: -1

Body: true

In database:



Advantages of this design

In my earlier articles I focussed more on design flaws, but our current design have emerged with few added advantages,

1. We got an extensible and loosely coupled application design that can go far with more new components added in the same way.
2. Registering types automatically through reflection. Suppose we want to register any Interface implementation to our REST endpoint, we just need to load that dll in our Bootstrapper class, or if dll's are of common suffix names then we just have to place that dll in bin folder, and that will automatically be loaded at run time.



Image source: http://a.pragprog.com/magazines/2013-07/images/iStock_000021011143XSmall_1mfk9b_.jpg

3. Database transactions or any of such module is now not exposed to the service endpoint, this makes our service more secure and maintains the design structure too.

Conclusion

We now know how to use Unity container to resolve dependency and perform inversion of control using MEF too. In the next chapter I'll try to explain how we can open multiple endpoints to our REST service and create custom url's in the true REST fashion in my WebAPI. Till then Happy Coding 😊 You can also download the source code from [GitHub](#). Add the required packages, if they are missing in the source code.

Custom URL Re-Writing/Routing Using Attribute Routes in MVC 4 Web APIs

Routing

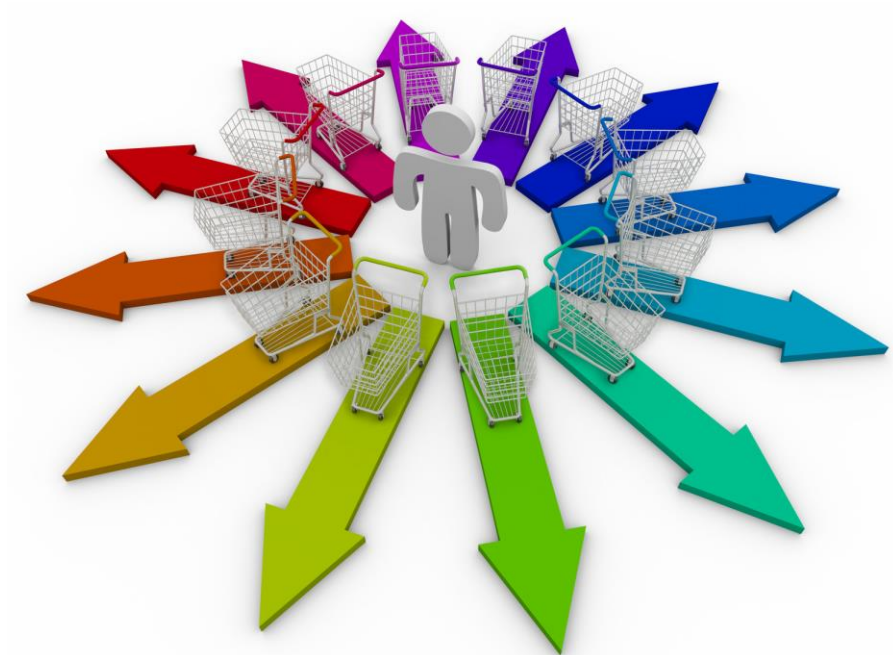
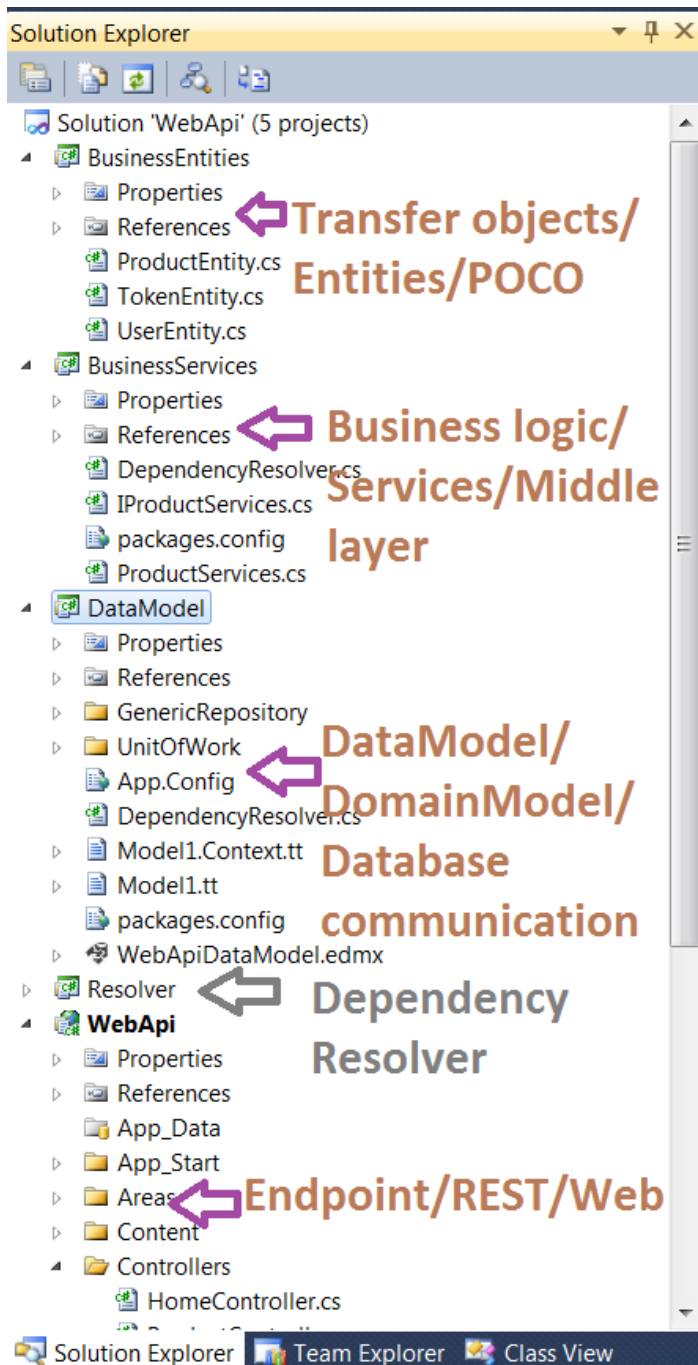


Image credit : <http://www.victig.com/wp-content/uploads/2010/12/routing.jpg>

Routing in generic terms for any service, api, website is a kind of pattern defining system that tries to map all request from the clients and resolves that request by providing some response to that request. In WebAPI we can define routes in WebAPIConfig file, these routes are defined in an internal Route Table. We can define multiple sets of Routes in that table.

Existing Design and Problem

We already have an existing design. If you open the solution, you'll get to see the structure as mentioned below,



In our existing application, we created WebAPI with default routes as mentioned in the file named WebApiConfig in App_Start folder of WebApi project. The routes were mentioned in the Register method as,

```
config.Routes.MapHttpRoute(
    name: "DefaultApi",
```



```
routeTemplate: "api/{controller}/{id}",  
defaults: new { id = RouteParameter.Optional }  
);
```

Do not get confused by MVC routes, since we are using MVC project we also get MVC routes defined in RouteConfig.cs file as,

```
public static void RegisterRoutes(RouteCollection routes)  
{  
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");  
  
    routes.MapRoute(  
        name: "Default",  
        url: "{controller}/{action}/{id}",  
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }  
    );  
}
```

We need to focus on the first one i.e. WebAPI route. As you can see in the following image, what each property signifies,


```

using System.Linq;
using System.Web.Http;

namespace WebApi
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}

```

We have a route name, we have a common URL pattern for all routes, and option to provide optional parameters as well.

Since our application do not have particular different action names, and we were using HTTP VERBS as action names, we didn't bother much about routes. Our Action names were like ,

1. `public HttpResponseMessage Get()`
2. `public HttpResponseMessage Get(int id)`
3. `public int Post([FromBody] ProductEntity productEntity)`
4. `public bool Put(int id, [FromBody] ProductEntity productEntity)`
5. `public bool Delete(int id)`

Default route defined does not takes HTTP VERBS action names into consideration and treat them default actions, therefore it does not mentions {action} in routeTemplate. But that's not limitation, we can have our own routes defined in WebApiConfig , for example, check out the following routes,


```

public static void Register(HttpConfiguration config)
{
    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    config.Routes.MapHttpRoute(
        name: "ActionBased",
        routeTemplate: "api/{controller}/{action}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
    config.Routes.MapHttpRoute(
        name: "ActionBased",
        routeTemplate: "api/{controller}/action/{action}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}

```

```

public static void Register(HttpConfiguration config)
{
    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );

    config.Routes.MapHttpRoute(
        name: "ActionBased",
        routeTemplate: "api/{controller}/{action}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}

```



```
config.Routes.MapHttpRoute(  
    name: "ActionBased",  
    routeTemplate: "api/{controller}/action/{action}/{id}",  
    defaults: new { id = RouteParameter.Optional }  
);  
}
```

And etc. In above mentioned routes, we can have action names as well, if we have custom actions.

So there is no limit to define routes in WebAPI. But there are few limitations to this, note we are talking about WebAPI 1 that we use with .Net Framework 4.0 in Visual Studio 2010. Web API 2 has overcome those limitations by including the solution that I'll explain in this article. Let's check out the limitations of these routes,



Every Controller will have to follow the same common route



The routes will be common to each and every action that we add to our controller



We can not have more than one endpoints to our service, and we'll be forced to stick to only one route



We can not have custom names of the routes. Routes depend on names of controller and action



We can not change the routes once they are defined, unless we agree to change the name of controllers and actions



Two or more action routes may conflict with each other, hence the priority will be given to the first route defined.

Yes, these are the limitations that I am talking about in Web API 1.

If we have route template like routeTemplate: "api/{controller}/{id}" or routeTemplate: "api/{controller}/{action}/{id}" or routeTemplate: "api/{controller}/action/{action}/{id}",

We can never have custom routes and will have to stick to old route convention provided by MVC. Suppose your client of the project wants to expose multiple endpoints for the service, he can't do so. We also can not have our own defined names for the routes, so lots of limitation.

Let's suppose we want to have following kind of routes for my web api endpoints, where I can define versions too,

v1/Products/Product/allproducts

v1/Products/Product/productid/1

v1/Products/Product/particularproduct/4

v1/Products/Product/myproduct/<with a range>

v1/Products/Product/create

v1/Products/Product/update/3

and so on, then we cannot achieve this with existing model.

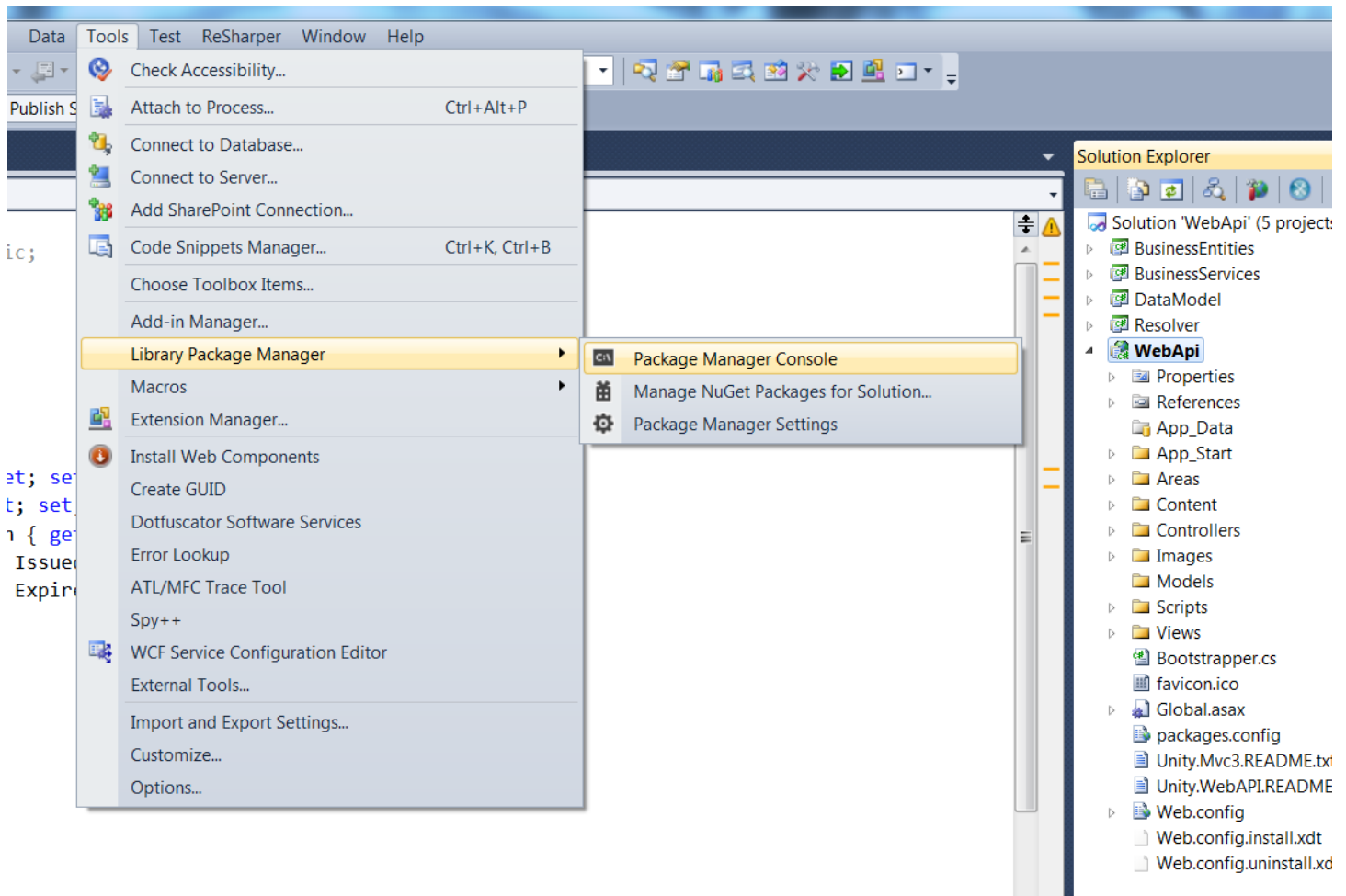
Fortunately these things have been taken care of in WebAPI 2 with MVC 5 , but for this situation we have `AttributeRouting` to resolve and overcome these limitations.

Attribute Routing

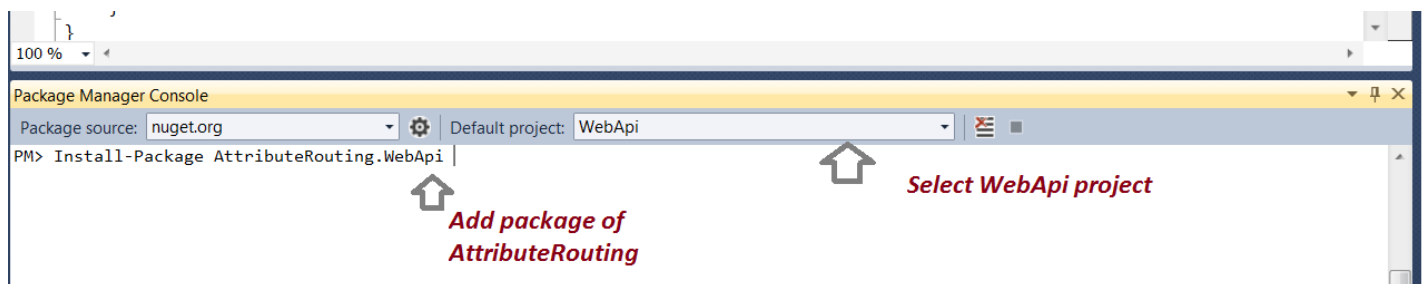
Attribute Routing is all about creating custom routes at controller level, action level. We can have multiple routes using Attribute Routing. We can have versions of routes as well, in short we have the solution for our exiting problems. Let's straight away jump on how we can implement this in our existing project. I am not explaining on how to create a WebAPI, for that you can refer my first [post](#) of the series.

Step 1: Open the solution ,and open Package Manage Console like shown below in the figure,

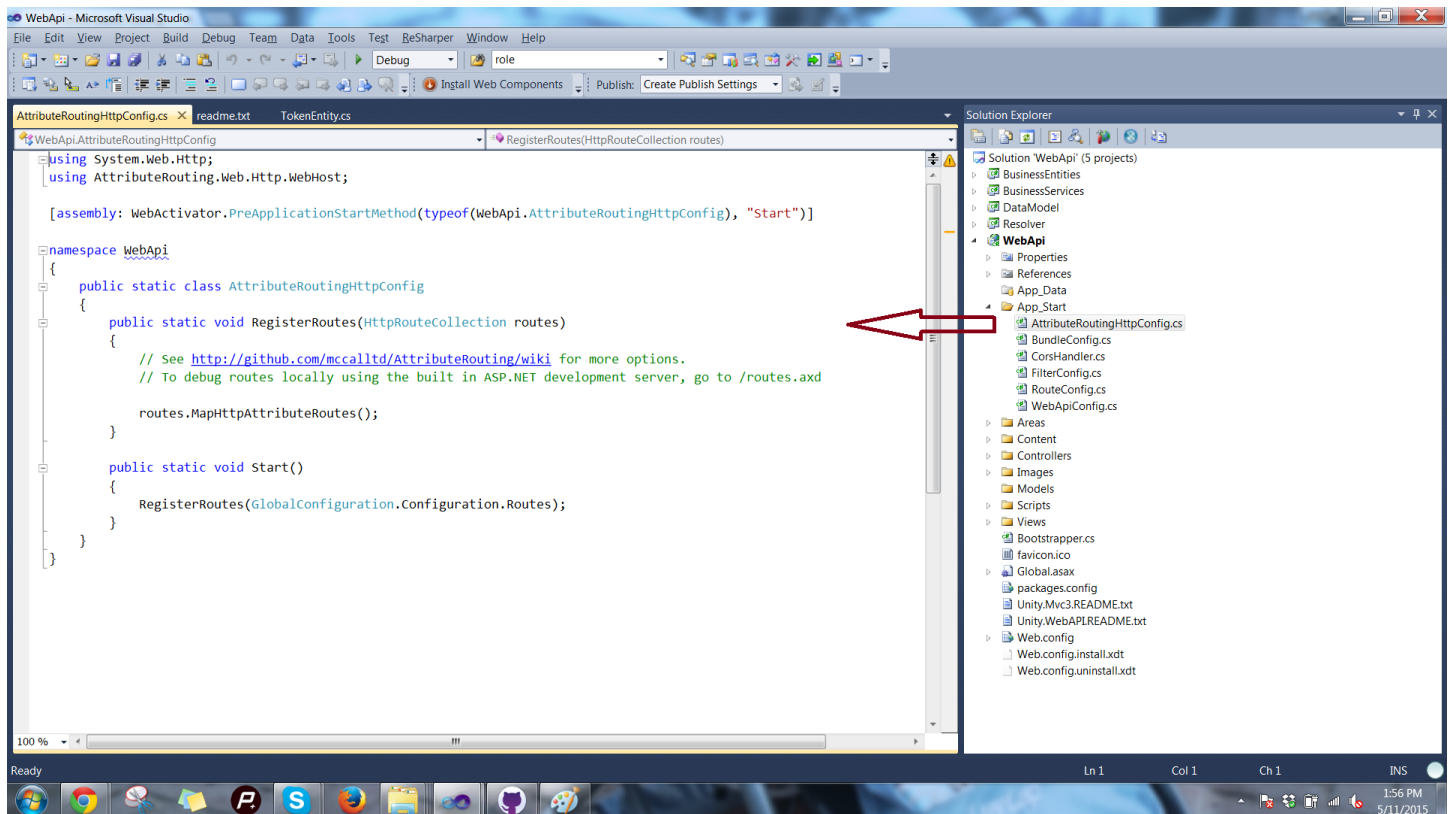
Goto Tools->Library Packet manage-> Packet Manager Console



Step 2: In the package manager console window at left corner of Visual Studio type, Install-Package AttributeRouting.WebApi, and choose the project WebApi or your own API project if you are using any other code sample, then press enter.



Step 3: As soon as the package is installed, you'll get a class named AttributeRoutingHttpConfig.cs in your App_Start folder.



This class has its own method to RegisterRoutes, which internally maps attribute routes. It has a start method that picks Routes defined from GlobalConfiguration and calls the RegisterRoutes method,

```
using System.Web.Http;
```

```
using AttributeRouting.Web.Http.WebHost;
```

```
[assembly:
```

```
WebActivator.PreApplicationStartMethod(typeof(WebApi.AttributeRoutingHttpConfig),  
"Start")]
```

```
namespace WebApi
```

```
{
```

```
    public static class AttributeRoutingHttpConfig
```

```
    {
```



```

    public static void RegisterRoutes(HttpRouteCollection routes)
    {
        // See http://github.com/mccalltd/AttributeRouting/wiki for more
options.

        // To debug routes locally using the built in ASP.NET development
server, go to /routes.axd

        routes.MapHttpAttributeRoutes();
    }

    public static void Start()
    {
        RegisterRoutes(GlobalConfiguration.Configuration.Routes);
    }
}

```

We don't even have to touch this class, our custom routes will automatically be taken care of using this class. We just need to focus on defining routes. No coding 😊. You can now use route specific stuff like route names, verbs, constraints, optional parameters, default parameters, methods, route areas, area mappings, route prefixes, route conventions etc.

Setup REST endpoint / WebAPI project to define Routes

Our 90% of the job is done.



We now need to setup or WebAPI project and define our routes.

Our existing ProductController class looks something like shown below,

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

```
using System.Linq;
```

```
using System.Net;
```

```
using System.Net.Http;
```

```
using System.Web.Http;
```

```
using BusinessEntities;
```

```
using BusinessServices;
```

```
namespace WebApi.Controllers
```

```
{
```

```
    public class ProductController : ApiController
```

```
    {
```



```
private readonly IProductServices _productServices;
```

```
#region Public Constructor
```

```
/// <summary>
```

```
/// Public constructor to initialize product service instance
```

```
/// </summary>
```

```
public ProductController(IProductServices productServices)
```

```
{
```

```
    _productServices = productServices;
```

```
}
```

```
#endregion
```

```
// GET api/product
```

```
public HttpResponseMessage Get()
```

```
{
```

```
    var products = _productServices.GetAllProducts();
```

```
    var productEntities = products as List<ProductEntity> ?? products.ToList();
```

```
    if (productEntities.Any())
```

```
        return Request.CreateResponse(HttpStatusCode.OK, productEntities);
```

```
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not  
found");
```



```
}
```

```
// GET api/product/5
```

```
public HttpResponseMessage Get(int id)
```

```
{
```

```
    var product = _productServices.GetProductById(id);
```

```
    if (product != null)
```

```
        return Request.CreateResponse(HttpStatusCode.OK, product);
```

```
    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found  
for this id");
```

```
}
```

```
// POST api/product
```

```
public int Post([FromBody] ProductEntity productEntity)
```

```
{
```

```
    return _productServices.CreateProduct(productEntity);
```

```
}
```

```
// PUT api/product/5
```

```
public bool Put(int id, [FromBody] ProductEntity productEntity)
```

```
{
```

```
    if (id > 0)
```

```
{
```



```
        return _productServices.UpdateProduct(id, productEntity);
    }

    return false;
}

// DELETE api/product/5

public bool Delete(int id)
{
    if (id > 0)
        return _productServices.DeleteProduct(id);

    return false;
}
}
```

Where we have a controller name **Product** and Action names as Verbs. When we run the application, we get following type of endpoints only (please ignore port and localhost setting. It's because I run this application from my local environment),

ASP.NET Web API Help Page

Introduction

Provide a general description of your APIs here.

Product

API	Description
GET api/Product	Documentation for 'Get'.
GET api/Product/{id}	Documentation for 'Get'.
POST api/Product	Documentation for 'Post'.
PUT api/Product/{id}	Documentation for 'Put'.
DELETE api/Product/{id}	Documentation for 'Delete'.

Get All Products:

<http://localhost:40784/api/Product>

Get product By Id:

<http://localhost:40784/api/Product/3>

Create product :

<http://localhost:40784/api/Product> (with json body)

Update product:

<http://localhost:40784/api/Product/3> (with json body)

Delete product:

<http://localhost:40784/api/Product/3>

Step 1: Add two namespaces to your controller,

`using` AttributeRouting;

using AttributeRouting.Web.Http;

Step 2: Decorate your action with different routes,

```

using AttributeRouting;
using AttributeRouting.Web.Http;
using BusinessEntities;
using BusinessServices;

namespace WebApi.Controllers
{
    public class ProductController : ApiController
    {
        private readonly IProductServices _productServices;

        // GET api/product
        public HttpResponseMessage Get()
        {
            var products = _productServices.GetAllProducts();
            var productEntities = products as List<ProductEntity> ?? products.ToList();
            if (productEntities.Any())
                return Request.CreateResponse(HttpStatusCode.OK, productEntities);
            return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products no");
        }

        // GET api/product/5
        [GET("productid/{id?}")]
        public HttpResponseMessage Get(int id)
        {
            var product = _productServices.GetProductById(id);
            if (product != null)
                return Request.CreateResponse(HttpStatusCode.OK, product);
        }
    }
}

```

Add using's

Define a new route to your Get method

Like in above image, I defined a route with name productid which taked id as a parameter. We also have to provide verb (GET, POST, PUT, DELETE, PATCH) along with the route like shown in image. So it is [GET("productid/{id?}")] . You can define whatever route

you want for your Action like [GET("product/id/{id?}")], [GET("myproduct/id/{id?}")] and many more.


Now when I run the application and navigate to /help page, I get this,

ASP.NET Web API Help Page

Introduction

Provide a general description of your APIs here.

Product

API	Description
GET productid	Documentation for 'Get'.
GET productid/{id}  One new route for get by id	Documentation for 'Get'.
GET api/Product	Documentation for 'Get'.
GET api/Product/{id}	Documentation for 'Get'.

i.e. I got one more route for Getting product by id. When you'll test this service you'll get your desired url something like : <http://localhost:55959/Product/productid/3>, that sounds like real REST 😊.

Similarly decorate your Action with multiple routes like show below,

```
// GET api/product/5

[GET("productid/{id?}")]

[GET("particularproduct/{id?}")]

[GET("myproduct/{id:range(1, 3)}")]

public HttpResponseMessage Get(int id)

{
```



```
var product = _productServices.GetProductById(id);

if (product != null)

    return Request.CreateResponse(HttpStatusCode.OK, product);

return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found
for this id");

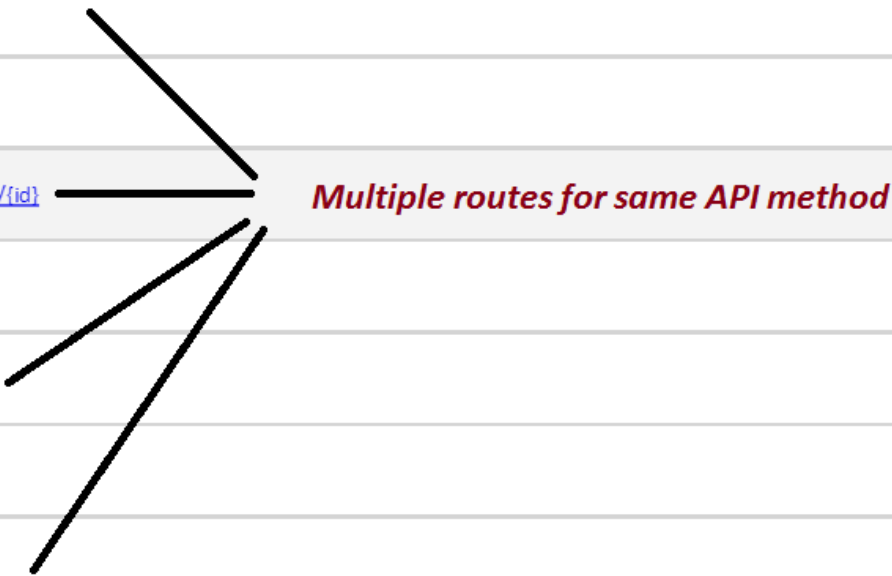
}
```

introduction

Provide a general description of your APIs here.

Product

API
GET myproduct
GET myproduct/{id}
GET particularproduct
GET particularproduct/{id}
GET productid
GET productid/{id}
GET api/Product
GET api/Product/{id}
POST api/Product
PUT api/Product/{id}



Therefore we see, we can have our custom route names and as well as multiple endpoints for a single Action. That's exciting. Each endpoint will be different but will serve same set of result.

- {id?} : here '?' means that parameter can be optional.
- [GET("myproduct/{id:range(1, 3)}")], signifies that the product id's falling in this range will only be shown.

More Routing Constraints

You can leverage numerous Routing Constraints provided by Attribute Routing. I am taking example of some of them,

Range:

To get the product within range, we can define the value, on condition that it should exist in database.

```
[GET("myproduct/{id:range(1, 3)}")]

public HttpResponseMessage Get(int id)
{
    var product = _productServices.GetProductById(id);

    if (product != null)
        return Request.CreateResponse(HttpStatusCode.OK, product);

    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found for this id");
}
```


Regular Expression:

You can use it well for text/string parameters more efficiently.

```
[GET(@"id/{e:regex(^[0-9]$)}")]
```

```

public HttpResponseMessage Get(int id)
{
    var product = _productServices.GetProductById(id);

    if (product != null)

        return Request.CreateResponse(HttpStatusCode.OK, product);

    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found
for this id");
}

```

e.g. [GET(@"text/{e:regex(^[A-Z][a-z][0-9]\$)}")]

Optional Parameters and Default Parameters:

You can also mark the service parameters as optional in the route. For example you want to fetch an employee detail from the data base with his name,

```
[GET("employee/name/{firstname}/{lastname?}")]
```

```

public string GetEmployeeName(string firstname, string lastname="mittal")
{

```



```
.....  
.....  
}
```

In the above mentioned code, I marked last name as optional by using question mark '?' to fetch the employee detail. It's my end user wish to provide the last name or not.

So the above endpoint could be accessed through GET verb with urls,

~/employee/name/akhil/mittal

~/employee/name/akhil

If a route parameter defined is marked optional, you must also provide a default value for that method parameter.

In the above example, I marked 'lastname' as an optional one and so provided a default value in the method parameter, if user doesn't send any value, "mittal" will be used.

In .Net 4.5 Visual Studio > 2010 with WebAPI 2, you can define DefaultRoute as an attribute too, just try it by your own. Use attribute [DefaultRoute] to define default route values.

You can try giving custom routes to all your controller actions.

I marked my actions as,


```
// GET api/product

[GET("allproducts")]

[GET("all")]

public HttpResponseMessage Get()
{
    var products = _productServices.GetAllProducts();

    var productEntities = products as List<ProductEntity> ?? products.ToList();

    if (productEntities.Any())

        return Request.CreateResponse(HttpStatusCode.OK, productEntities);

    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not
found");
}

// GET api/product/5

[GET("productid/{id?}")]

[GET("particularproduct/{id?}")]

[GET("myproduct/{id:range(1, 3)}")]

public HttpResponseMessage Get(int id)
{
    var product = _productServices.GetProductById(id);

    if (product != null)

        return Request.CreateResponse(HttpStatusCode.OK, product);
}
```



```
        return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found  
for this id");
```

```
    }
```

```
// POST api/product
```

```
[POST("Create")]
```

```
[POST("Register")]
```

```
public int Post([FromBody] ProductEntity productEntity)
```

```
{
```

```
    return _productServices.CreateProduct(productEntity);
```

```
}
```

```
// PUT api/product/5
```

```
[PUT("Update/productid/{id}")]
```

```
[PUT("Modify/productid/{id}")]
```

```
public bool Put(int id, [FromBody] ProductEntity productEntity)
```

```
{
```

```
    if (id > 0)
```

```
    {
```

```
        return _productServices.UpdateProduct(id, productEntity);
```

```
    }
```

```
    return false;
```

```
}
```



```
// DELETE api/product/5  
  
[DELETE("remove/productid/{id}")]  
  
[DELETE("clear/productid/{id}")]  
  
[PUT("delete/productid/{id}")]  
  
public bool Delete(int id)  
{  
    if (id > 0)  
        return _productServices.DeleteProduct(id);  
    return false;  
}
```

And therefore get the routes as,

GET:

Product

API	Description
GET v1/Products/Product/all	Documentation for 'Get'.
GET v1/Products/Product/all?id={id}	Documentation for 'Get'.
GET v1/Products/Product/allproducts	Documentation for 'Get'.
GET v1/Products/Product/allproducts?id={id}	Documentation for 'Get'.
GET v1/Products/Product/myproduct/{id}	Documentation for 'Get'.
GET v1/Products/Product/particularproduct	Documentation for 'Get'.
GET v1/Products/Product/particularproduct/{id}	Documentation for 'Get'.
GET v1/Products/Product/productid	Documentation for 'Get'.
GET v1/Products/Product/productid/{id}	Documentation for 'Get'.

POST / PUT / DELETE:

POST v1/Products/Product/Register	Documentation for 'Post'.
POST v1/Products/Product/Create	Documentation for 'Post'.
PUT v1/Products/Product/Update/productid/{id}	Documentation for 'Put'.
PUT v1/Products/Product/Modify/productid/{id}	Documentation for 'Put'.
DELETE v1/Products/Product/delete/productid/{id}	Documentation for 'Delete'.
DELETE v1/Products/Product/remove/productid/{id}	Documentation for 'Delete'.
DELETE v1/Products/Product/clear/productid/{id}	Documentation for 'Delete'.
GET api/Product	Documentation for 'Get'.
GET api/Product/{id}	Documentation for 'Get'.
POST api/Product	Documentation for 'Post'.
PUT api/Product/{id}	Documentation for 'Put'.

Check for more constraints [here](#).

You must be seeing “**v1/Products**” in every route, that is due to RoutePrefix I have used at controller level. Let's discuss RoutePrefix in detail.

RoutePrefix: Routing at Controller level

We were marking our actions with particular route, but guess what, we can mark our controllers too with certain route names, we can achieve this by using RoutePrefix attribute of AttributeRouting. Our controller was named Product, and I want to append

Products/Product before my every action, there fore without duplicating the code at each and every action, I can decorate my Controller class with this name like shown below,

```
[RoutePrefix("Products/Product")]

public class ProductController : ApiController

{
```

Now, since our controller is marked with this route, it will append the same to every action too. For e.g. route of following action,

```
[GET("allproducts")]

[GET("all")]

public HttpResponseMessage Get()

{

    var products = _productServices.GetAllProducts();

    var productEntities = products as List<ProductEntity> ?? products.ToList();

    if (productEntities.Any())

        return Request.CreateResponse(HttpStatusCode.OK, productEntities);

    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not
found");

}
```

Now becomes,

~/Products/Product/allproducts

~/Products/Product/all

RoutePrefix: Versioning

Route prefix can also be used for versioning of the endpoints, like in my code I provided “v1” as version in my RoutePrefix as shown below,

```
[RoutePrefix("v1/Products/Product")]  
  
public class ProductController : ApiController  
{
```

Therefore “v1” will be appended to every route / endpoint of the service. When we release next version, we can certainly maintain a change log separately and mark the endpoint as “v2” at controller level, that will append “v2” to all actions,

e.g.

~/v1/Products/Product/allproducts

~/v1/Products/Product/all

~/v2/Products/Product/allproducts

~/v2/Products/Product/all

RoutePrefix: Overriding

This functionality is present in .Net 4.5 with Visual Studio > 2010 with WebAPI 2. You can test it there.

There could be situations that we do not want to use RoutePrefix for each and every action. AttributeRouting provides such flexibility too, that despite of RoutePrefix present at controller level, an individual action could have its own route too. It just needs to override the default route like shown below,

RoutePrefix at Controller :

```
[RoutePrefix("v1/Products/Product")]  
  
public class ProductController : ApiController  
{
```

Independent Route of Action:

```
[Route("~/MyRoute/allproducts")]  
  
public HttpResponseMessage Get()  
{  
  
    var products = _productServices.GetAllProducts();  
  
    var productEntities = products as List<ProductEntity> ?? products.ToList();  
  
    if (productEntities.Any())
```



```
        return Request.CreateResponse(HttpStatusCode.OK, productEntities);

        return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not
found");
    }
```

Disable Default Routes

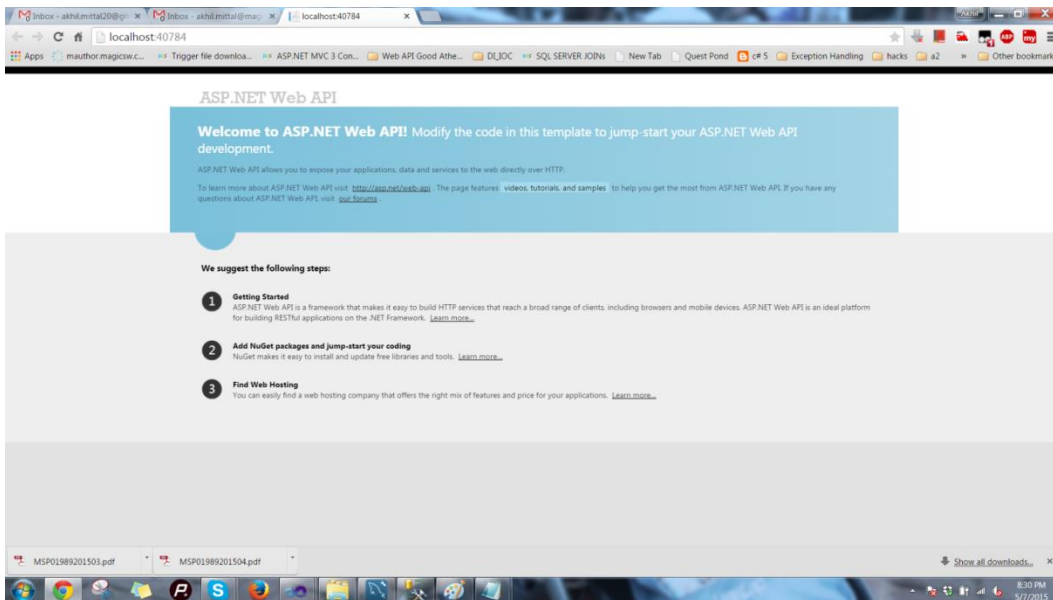
You must be wondering that in the list of all the URL's on service help page, we are getting some different/other routes that we have not defined through attribute routing starting like ~/api/Product. These routes are the outcome of default route provided in [WebApiConfig](#) file, remember??. If you want to get rid of those unwanted routes, just go and comment everything written in Register method in [WebApiConfig.cs](#) file under App_Start folder,

```
//config.Routes.MapHttpRoute(
//    name: "DefaultApi",
//    routeTemplate: "api/{controller}/{id}",
//    defaults: new { id = RouteParameter.Optional }
//);
```

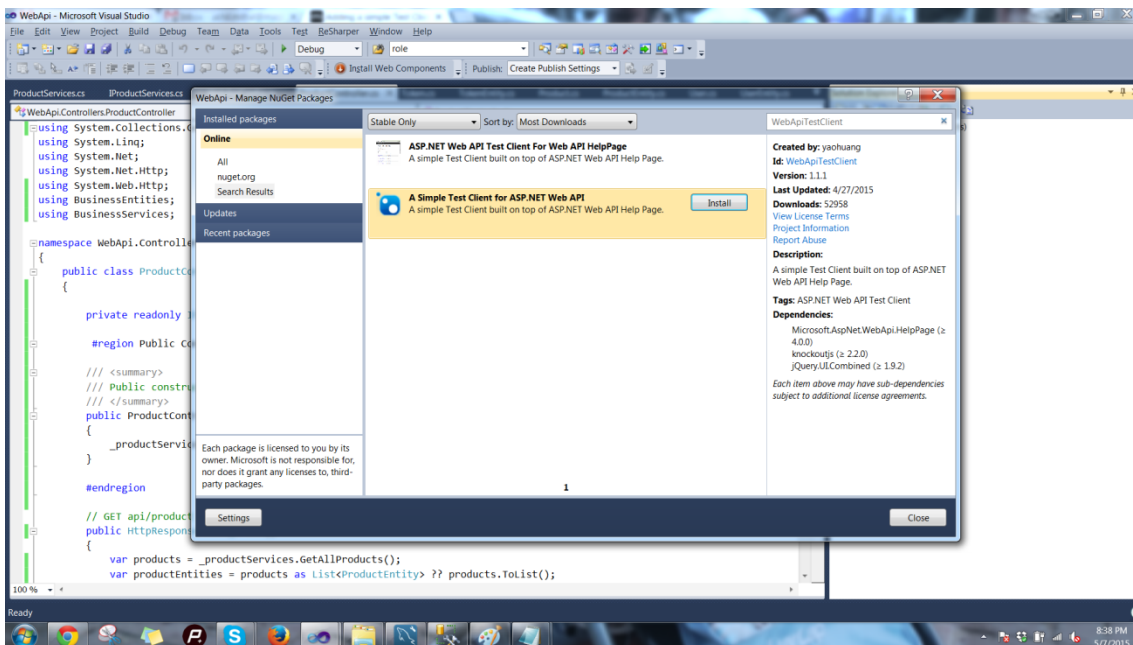
You can also remove the complete Register method, but for that you need to remove its calling too from Global.asax file.

Running the application

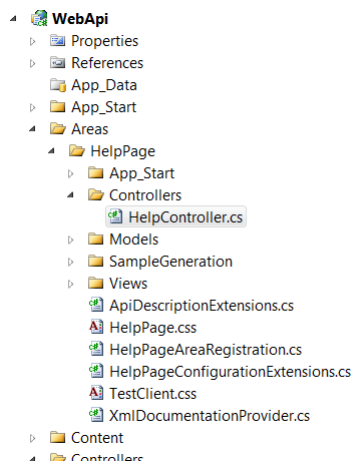
Just run the application, we get,



We already have our test client added, but for new readers, just go to Manage Nuget Packages, by right clicking WebAPI project and type WebAPITestClient in searchbox in online packages,



You'll get "A simple Test Client for ASP.NET Web API", just add it. You'll get a help controller in Areas-> HelpPage like shown below,



I have already provided the database scripts and data in my previous article, you can use the same.

Append “/help” in the application url, and you’ll get the test client,

GET:

Product

API	Description
GET v1/Products/Product/all	Documentation for 'Get'.
GET v1/Products/Product/all?id={id}	Documentation for 'Get'.
GET v1/Products/Product/allproducts	Documentation for 'Get'.
GET v1/Products/Product/allproducts?id={id}	Documentation for 'Get'.
GET v1/Products/Product/myproduct/{id}	Documentation for 'Get'.
GET v1/Products/Product/particularproduct	Documentation for 'Get'.
GET v1/Products/Product/particularproduct/{id}	Documentation for 'Get'.
GET v1/Products/Product/productid	Documentation for 'Get'.
GET v1/Products/Product/productid/{id}	Documentation for 'Get'.

POST:

POST v1/Products/Product/Create	Documentation for 'Post'.
POST v1/Products/Product/Register	Documentation for 'Post'.

PUT:

[PUT v1/Products/Product/Update/productid/{id}](#)

Documentation for 'Put'.

[PUT v1/Products/Product/Modify/productid/{id}](#)

Documentation for 'Put'.

DELETE:

[DELETE v1/Products/Product/delete/productid/{id}](#)

Documentation for 'Delete'.

[DELETE v1/Products/Product/remove/productid/{id}](#)

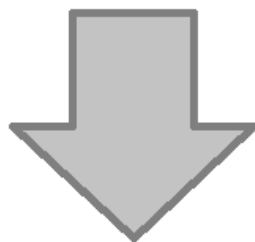
Documentation for 'Delete'.

[DELETE v1/Products/Product/clear/productid/{id}](#)

Documentation for 'Delete'.

You can test each service by clicking on it. Once you click on the service link, you'll be redirected to test the service page of that particular service. On that page there is a button Test API in the right bottom corner, just press that button to test your service,

**Press this button to
test the API**



A rectangular button with a light gray background and a thin black border. The text 'Test API' is centered on the button in a black, sans-serif font.

Service for Get All products,

[Help](#) [Page](#) [Home](#)

GET v1/Products/Product/all

Documentation for 'Get'.

GET v1/Products/Product/all x

GET

Headers | [Add header](#)

☐ **Body**

Send

Response for GET v1/Products/Product/all x

Status

200/OK

Headers

Server: ASP.NET Development Server/10.0.0.0
Date: Sun, 31 May 2015 09:11:10 GMT
X-AspNet-Version: 4.0.30319
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Content-Type: application/json; charset=utf-8

Body

```
{
  "ProductId": 1,
  "ProductName": "Laptop"
},
{
  "ProductId": 2,
  "ProductName": "computer"
},
{
  "ProductId": 4,
  "ProductName": "IPhone"
}
}
```


Likewise you can test all the service endpoints.

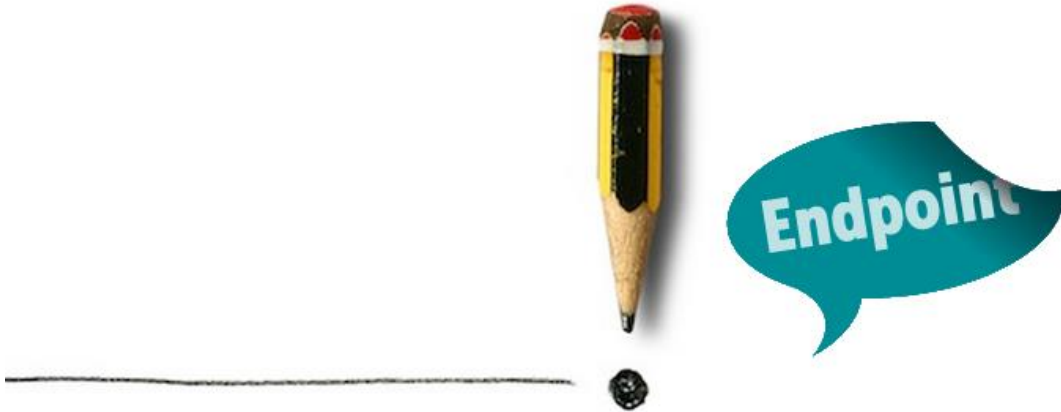


Image source: <http://www.rsc.org/eic/sites/default/files/upload/cwtype-Endpoint.jpg> and http://www.guybutlerphotography.com/EEGDemoConstruction/wp-content/uploads/2013/01/asthma-endpoint_shutterstock_89266522_small.jpg

Conclusion

We now know how to define our custom endpoints and what its benefits are. Just to share that this library was introduced by Tim Call, author of <http://attributerouting.net> and Microsoft has included this in WebAPI 2 by default. My next article will explain token based authentication using ActionFilters in WepAPI. Till then Happy Coding 😊 you can also download the complete source code from [GitHub](#). Add the required packages, if they are missing in the source code.

Click [Github Repository](#) to browse for complete source code.

References

<http://blogs.msdn.com/b/webdev/archive/2013/10/17/attribute-routing-in-asp-net-mvc-5.aspx>

<https://github.com/mccalltd/AttributeRouting>

Security in Web API - Basic & Token Based Custom Authorization in Web APIs Using Action Filters

Security in WebAPI

Security in itself is very complicated and tricky topic. I'll try to explain how we can achieve it in WebAPI in my own way.

When we plan to create an enterprise level application, we especially want to take care of authentication and authorization. These are two techniques if used well makes our application secure, in our case makes our WebAPI more secure.



Image credit: https://pixabay.com/static/uploads/photo/2014/12/25/10/56/road-sign-579554_180.jpg

Authentication

Authentication is all about the identity of an end user. It's about validating the identity of a user who is accessing our system, that he is authenticated enough to use our resources or not. Does that end user have valid credentials to log in our system? Credentials can be in the form of a user name and password. We'll use Basic Authentication technique to understand how we can achieve authentication in WebAPI.

Authorization

Authorization should be considered as a second step after authentication to achieve security. Authorization means what all permissions the authenticated user has to access web resources. Is allowed to access/ perform action on that resource? This could be achieved by setting roles and permissions for an end user who is authenticated, or can be achieved through providing a secure token, using which an end user can have access to other services or resources.

Maintaining Session

RESTful services work on a stateless protocol i.e. HTTP. We can achieve maintaining session in Web API through token based authorization technique. An authenticated user will be allowed to access resources for a particular period of time, and can re-instantiate the request with an increased session time delta to access other resource or the same resource. Websites using WebAPIs as RESTful services may need to implement login/logout for a user, to maintain sessions for the user, to provide roles and permissions to their user, all these features could be achieved using basic authentication and token based authorization. I'll explain this step by step.

Basic Authentication

Basic authentication is a mechanism, where an end user gets authenticated through our service i.e. RESTful service with the help of plain credentials such as user name and password. An end user makes a request to the service for authentication with user name and password embedded in request header. Service receives the request and checks if the credentials are valid or not, and returns the response accordingly, in case of invalid credentials, service responds with 401 error code i.e. unauthorized. The actual credentials through which comparison is done may lie in database, any config file like web.config or in the code itself.

Pros and Cons of Basic Authentication

Basic authentication has its own pros and cons. It is advantageous when it comes to implementation, it is very easy to implement, it is nearly supported by all modern browsers and has become an authentication standard in RESTful / Web APIs. It has disadvantages like sending user credentials in plain text, sending user credentials inside request header, i.e. prone to hack. One has to send credentials each time a service is called. No session is maintained and a user cannot logout once logged in through basic authentication. It is very prone to CSRF (Cross Site Request Forgery).

Token Based Authorization

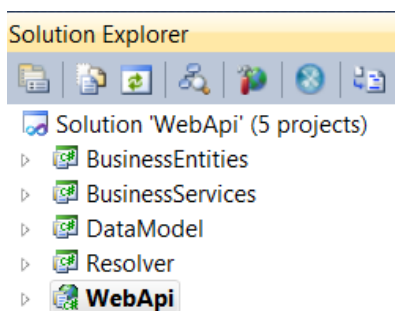
Authorization part comes just after authentication, once authenticated a service can send a token to an end user through which user can access other resources. The token could be any encrypted key, which only server/service understands and when it fetches the token from the request made by end user, it validates the token and authorizes user into the system. Token generated could be stored in a database or an external file as well i.e. we need to persist the token for future references. Token can have its own lifetime, and may expire accordingly. In that case user will again have to be authenticated into the system.

WebAPI with Basic Authentication and Token Based Authorization



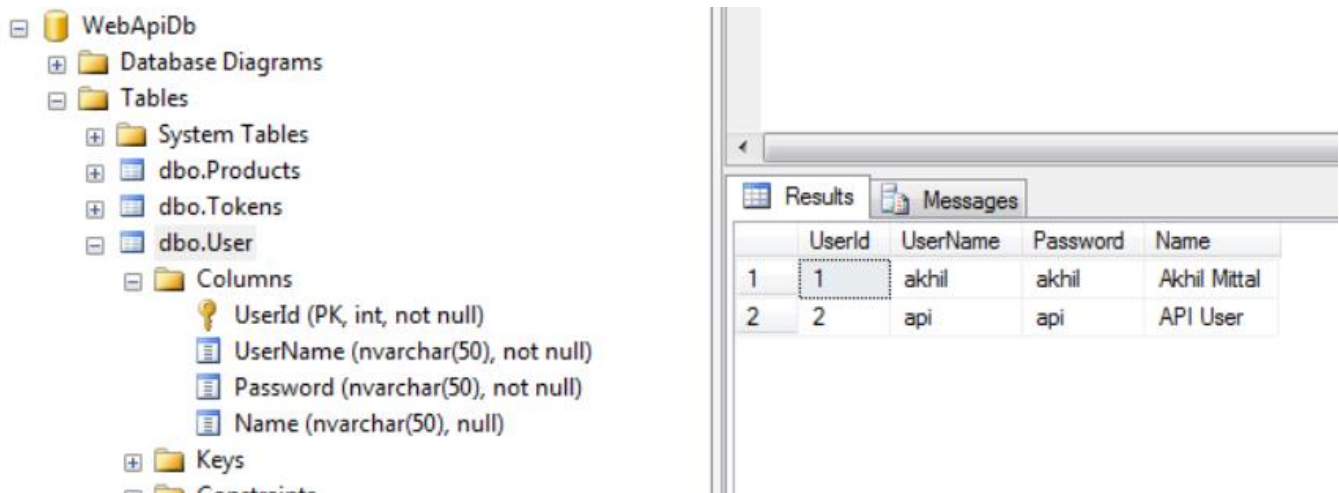
Creating User Service

Just open your WebAPI project or the WebAPI project that we discussed in the [last](#) part of learning WebAPI.



We have BusinessEntities, BusinessServices, DataModel, DependencyResolver and a WebApi project as well.

We already have a User table in database, or you can create your own database with a table like User Table as shown below,



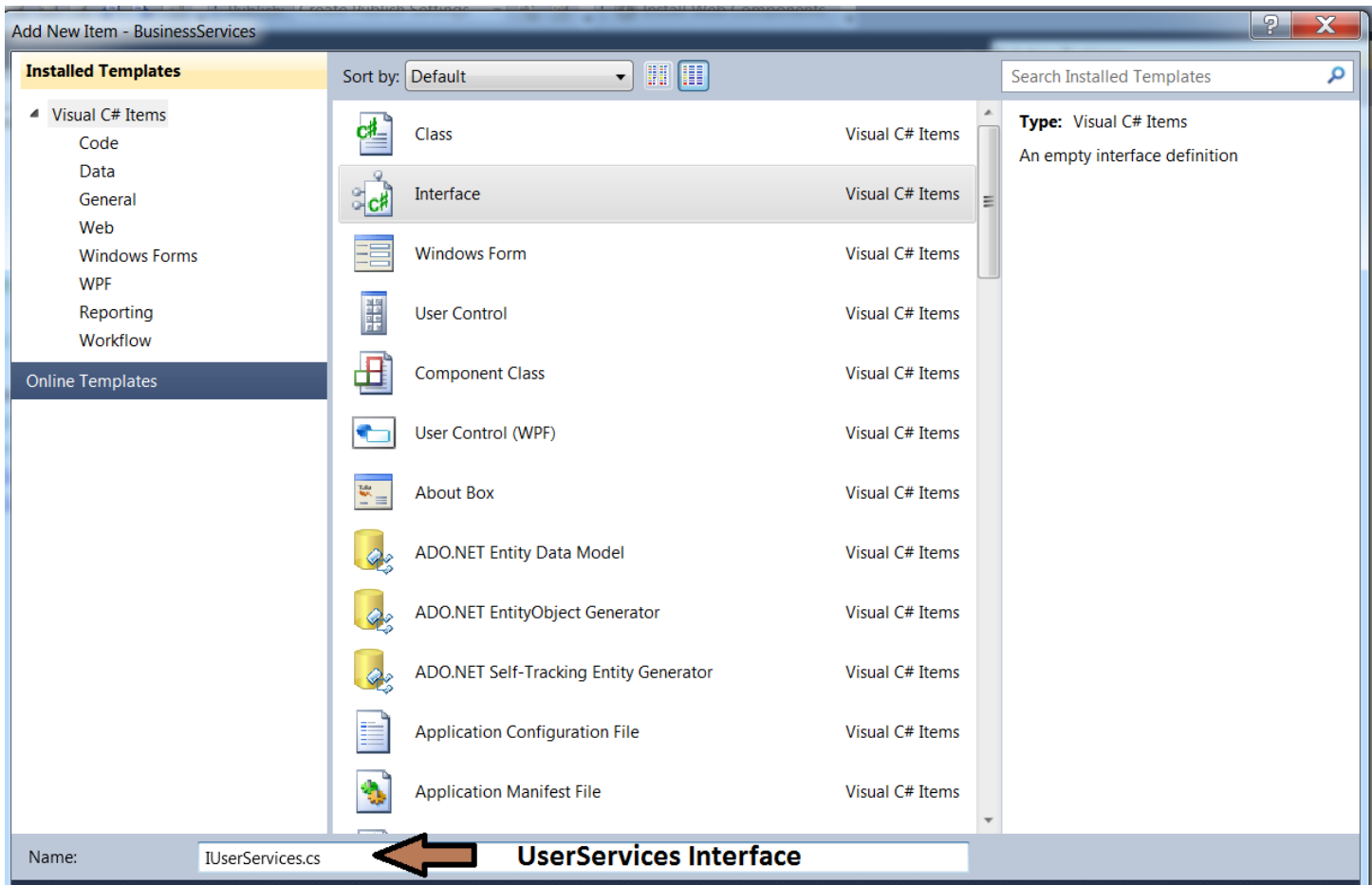
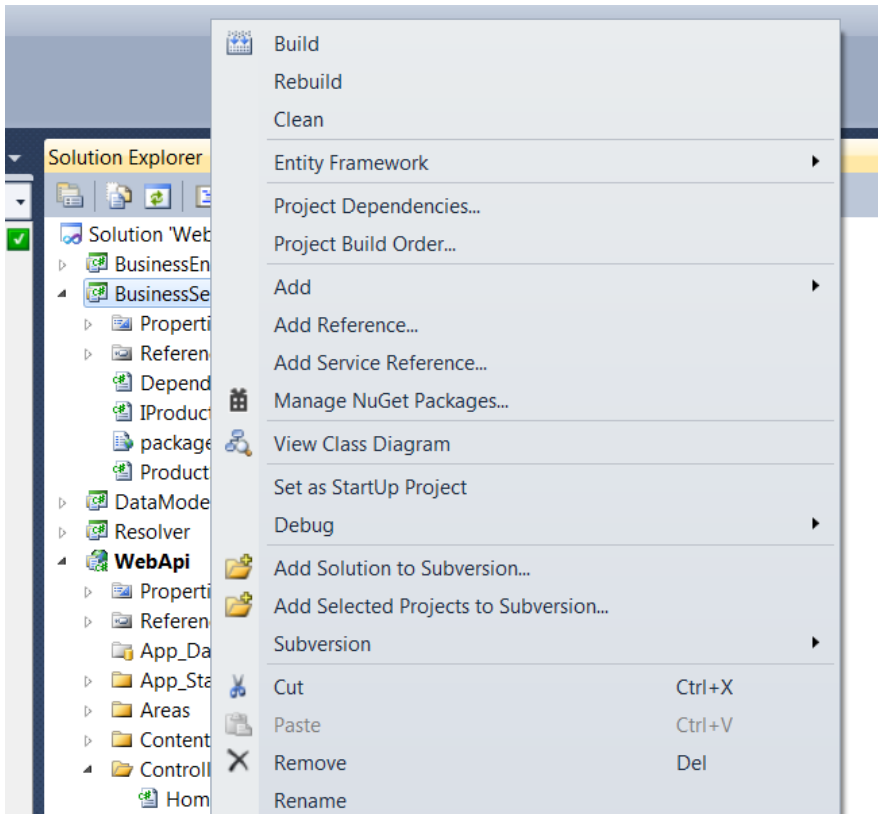
The screenshot shows the SQL Server Enterprise Manager interface. On the left, the 'WebApiDb' database is expanded, showing 'Database Diagrams', 'Tables', and 'Columns'. The 'dbo.User' table is selected, and its columns are listed: 'UserId' (PK, int, not null), 'UserName' (nvarchar(50), not null), 'Password' (nvarchar(50), not null), and 'Name' (nvarchar(50), null). On the right, the 'Results' tab is active, displaying a query result with 2 rows and 5 columns: 'UserId', 'UserName', 'Password', and 'Name'. The first row has values 1, akhil, akhil, and Akhil Mittal. The second row has values 2, api, api, and API User.

	UserId	UserName	Password	Name
1	1	akhil	akhil	Akhil Mittal
2	2	api	api	API User

I am using WebAPI database, scripts I have attached for download.

UserServices:

Go to BusinessServices project and add a new interface IUserService and a service named UserServices implementing that interface,



Just define one method named `Authenticate` in the interface.

```
namespace BusinessServices
{
    public interface IUserServices
    {
        int Authenticate(string userName, string password);
    }
}
```

This method takes username and password as a parameter and returns particular `userId` if the user is authenticated successfully.

Just implement this method in `UserServices.cs` class, just like we created services earlier in the series,

```
using DataModel.UnitOfWork;

namespace BusinessServices
{
    /// <summary>
    /// Offers services for user specific operations
```



```

/// </summary>

public class UserServices : IUserServices
{
    private readonly UnitOfWork _unitOfWork;

    /// <summary>
    /// Public constructor.
    /// </summary>

    public UserServices(UnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

    /// <summary>
    /// Public method to authenticate user by user name and password.
    /// </summary>
    /// <param name="userName"></param>
    /// <param name="password"></param>
    /// <returns></returns>

    public int Authenticate(string userName, string password)
    {
        var user = _unitOfWork.UserRepository.Get(u => u.UserName == userName &&
u.Password == password);

```



```
        if (user != null && user.UserId > 0)
        {
            return user.UserId;
        }

        return 0;
    }
}
```

You can clearly see that Authenticate method just checks user credentials from UserRepository and returns the values accordingly. The code is very much self-explanatory.

Resolve dependency of UserService:

Just open DependencyResolver class in BusinessServices project itself and add its dependency type so that we get UserServices dependency resolved at run time,so add

```
registerComponent.RegisterType<IUserServices, UserServices>();
```

line to SetUP method.Our class becomes,

```
using System.ComponentModel.Composition;
```

```
using DataModel;
```

```
using DataModel.UnitOfWork;
```

```
using Resolver;
```



```

namespace BusinessServices
{
    [Export(typeof(IComponent))]

    public class DependencyResolver : IComponent
    {
        public void SetUp(IRegisterComponent registerComponent)
        {
            registerComponent.RegisterType<IProductServices, ProductServices>();
            registerComponent.RegisterType<IUserServices, UserServices>();
        }
    }
}

```

Implementing Basic Authentication

Step 1: Create generic Authentication Filter

Add a folder named Filters to the WebAPI project and add a class named GenericAuthenticationFilter under that folder.

Derive that class from `AuthorizationFilterAttribute`, this is a class under `System.Web.Http.Filters`.

I have created the generic authentication filter will be like,


```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false)]
```

```
public class GenericAuthenticationFilter : AuthorizationFilterAttribute
```

```
{
```

```
    /// <summary>
```

```
    /// Public default Constructor
```

```
    /// </summary>
```

```
    public GenericAuthenticationFilter()
```

```
    {
```

```
    }
```

```
    private readonly bool _isActive = true;
```

```
    /// <summary>
```

```
    /// parameter isActive explicitly enables/disables this filter.
```

```
    /// </summary>
```

```
    /// <param name="isActive"></param>
```

```
    public GenericAuthenticationFilter(bool isActive)
```

```
    {
```

```
        _isActive = isActive;
```

```
    }
```



```
/// <summary>
/// Checks basic authentication request
/// </summary>
/// <param name="filterContext"></param>
public override void OnAuthorization(HttpActionContext filterContext)
{
    if (!_isActive) return;

    var identity = FetchAuthHeader(filterContext);

    if (identity == null)
    {
        ChallengeAuthRequest(filterContext);

        return;
    }

    var genericPrincipal = new GenericPrincipal(identity, null);
    Thread.CurrentPrincipal = genericPrincipal;

    if (!OnAuthorizeUser(identity.Name, identity.Password, filterContext))
    {
        ChallengeAuthRequest(filterContext);

        return;
    }

    base.OnAuthorization(filterContext);
}
```



```

/// <summary>
/// Virtual method.Can be overridden with the custom Authorization.
/// </summary>
/// <param name="user"></param>
/// <param name="pass"></param>
/// <param name="filterContext"></param>
/// <returns></returns>

```

```

protected virtual bool OnAuthorizeUser(string user, string pass, HttpContext
filterContext)

```

```

{
    if (string.IsNullOrEmpty(user) || string.IsNullOrEmpty(pass))
        return false;
    return true;
}

```

```

/// <summary>

```

```

/// Checks for authentication header in the request and parses it, creates user
credentials and returns as BasicAuthenticationIdentity

```

```

/// </summary>
/// <param name="filterContext"></param>

```

```

protected virtual BasicAuthenticationIdentity FetchAuthHeader(HttpContext
filterContext)

```

```

{
    string authHeaderValue = null;

```



```

var authRequest = filterContext.Request.Headers.Authorization;

if (authRequest != null && !String.IsNullOrEmpty(authRequest.Scheme) &&
authRequest.Scheme == "Basic")

    authHeaderValue = authRequest.Parameter;

if (string.IsNullOrEmpty(authHeaderValue))

    return null;

authHeaderValue =
Encoding.Default.GetString(Convert.FromBase64String(authHeaderValue));

var credentials = authHeaderValue.Split(':');

return credentials.Length < 2 ? null : new BasicAuthenticationIdentity(credentials[0],
credentials[1]);
}

```

```

/// <summary>
/// Send the Authentication Challenge request
/// </summary>
/// <param name="filterContext"></param>

private static void ChallengeAuthRequest(HttpActionContext filterContext)
{
    var dnsHost = filterContext.Request.RequestUri.DnsSafeHost;

    filterContext.Response =
filterContext.Request.CreateResponse(HttpStatusCode.Unauthorized);

    filterContext.Response.Headers.Add("WWW-Authenticate", string.Format("Basic
realm=\"{0}\"", dnsHost));
}

```



```

    }

}

```

Since this is an `AuthorizationFilter` derived class, we need to override its methods to add our custom logic. Here “`OnAuthorization`” method is overridden to add a custom logic. Whenever we get `ActionContext` on `OnAuthorization`, we’ll check for its header, since we are pushing our service to follow `BasicAuthentication`, the request headers should contain this information. I have used `FetchAuthHeader` to check the scheme, if it comes to be “`Basic`” and thereafter store the credentials i.e. user name and password in a form of an object of class `BasicAuthenticationIdentity`, therefore creating an identity out of valid credentials.

```

protected virtual BasicAuthenticationIdentity FetchAuthHeader(HttpContext
filterContext)
{
    string authHeaderValue = null;

    var authRequest = filterContext.Request.Headers.Authorization;

    if (authRequest != null && !String.IsNullOrEmpty(authRequest.Scheme) &&
authRequest.Scheme == "Basic")

        authHeaderValue = authRequest.Parameter;

    if (string.IsNullOrEmpty(authHeaderValue))

        return null;

    authHeaderValue =
Encoding.Default.GetString(Convert.FromBase64String(authHeaderValue));

    var credentials = authHeaderValue.Split(':');

    return credentials.Length < 2 ? null : new BasicAuthenticationIdentity(credentials[0],
credentials[1]);
}

```



```
}
```

I am expecting values to be encrypted using Base64 string; You can use your own encryption mechanism as well.

Later on in OnAuthorization method we create a genericPrincipal with the created identity and assign it to current Thread principal,

```
var genericPrincipal = new GenericPrincipal(identity, null);

Thread.CurrentPrincipal = genericPrincipal;

if (!OnAuthorizeUser(identity.Name, identity.Password, filterContext))
{
    ChallengeAuthRequest(filterContext);

    return;
}

base.OnAuthorization(filterContext);
```

Once done, a challenge to that request is added, where we add response and tell the Basic realm,

```
filterContext.Response.Headers.Add("WWW-Authenticate", string.Format("Basic realm=\"{0}\"", dnsHost));
```

in ChallengeAuthRequest method.

If no credentials is provided in the request, this generic authentication filter sets generic authentication principal to the current thread principal.

Since we know the drawback that in basic authentication credentials are passed in a plain text, so it would be good if our service uses SSL for communication or message passing.

We have an overridden constructor as well that allows to stop the default behavior of the filter by just passing in a parameter i.e. true or false.

```
public GenericAuthenticationFilter(bool isActive)
{
    _isActive = isActive;
}
```

We can use `OnAuthorizeUser` for custom authorization purposes.

Step 2: Create Basic Authentication Identity

Before we proceed further, we also need the `BasicIdentity` class, that takes hold of credentials and assigns it to Generic Principal. So just add one more class named `BasicAuthenticationIdentity` deriving from `GenericIdentity`.

This class contains three properties i.e. `UserName`, `Password` and `UserId`. I purposely added `UserId` because we'll need that in future. So our class will be like,

```
using System.Security.Principal;
```

```
namespace WebApi.Filters
```

```
{
    /// <summary>
```



```
/// Basic Authentication identity

/// </summary>

public class BasicAuthenticationIdentity : GenericIdentity
{
    /// <summary>
    /// Get/Set for password
    /// </summary>

    public string Password { get; set; }

    /// <summary>
    /// Get/Set for UserName
    /// </summary>

    public string UserName { get; set; }

    /// <summary>
    /// Get/Set for UserId
    /// </summary>

    public int UserId { get; set; }

    /// <summary>
    /// Basic Authentication Identity Constructor
    /// </summary>

    /// <param name="userName"></param>
    /// <param name="password"></param>

    public BasicAuthenticationIdentity(string userName, string password)
```



```

        : base(userName, "Basic")
    {
        Password = password;
        UserName = userName;
    }
}
}

```

Step 3: Create a Custom Authentication Filter

Now you are ready to use your own Custom Authentication filter. Just add one more class under that Filters project and call it `ApiAuthenticationFilter`, this class will derive from `GenericAuthenticationFilter`, that we created in first step. This class overrides `OnAuthorizeUser` method to add custom logic for authenticating a request. It makes use of `UserService` that we created earlier to check the user,

```

protected override bool OnAuthorizeUser(string username, string password,
HttpActionContext actionContext)
{
    var provider = actionContext.ControllerContext.Configuration
        .DependencyResolver.GetService(typeof(IUserServices)) as IUserServices;

    if (provider != null)
    {
        var userId = provider.Authenticate(username, password);

        if (userId > 0)
    }
}

```



```

    {
        var basicAuthenticationIdentity = Thread.CurrentPrincipal.Identity as
BasicAuthenticationIdentity;

        if (basicAuthenticationIdentity != null)

            basicAuthenticationIdentity.UserId = userId;

        return true;
    }
}

return false;
}

```

Complete class:

```

using System.Threading;

using System.Web.Http.Controllers;

using BusinessServices;

namespace WebApi.Filters
{
    /// <summary>
    /// Custom Authentication Filter Extending basic Authentication
    /// </summary>

    public class ApiAuthenticationFilter : GenericAuthenticationFilter

```



```

{

    /// <summary>
    /// Default Authentication Constructor
    /// </summary>

    public ApiAuthenticationFilter()

    {

    }

    /// <summary>
    /// AuthenticationFilter constructor with isActive parameter
    /// </summary>
    /// <param name="isActive"></param>

    public ApiAuthenticationFilter(bool isActive)
        : base(isActive)

    {

    }

    /// <summary>
    /// Protected overridden method for authorizing user
    /// </summary>
    /// <param name="username"></param>
    /// <param name="password"></param>
    /// <param name="actionContext"></param>

```



```

/// <returns></returns>

protected override bool OnAuthorizeUser(string username, string password,
HttpContext actionContext)
{
    var provider = actionContext.ControllerContext.Configuration
        .DependencyResolver.GetService(typeof(IUserServices)) as IUserServices;

    if (provider != null)
    {
        var userId = provider.Authenticate(username, password);

        if (userId > 0)
        {
            var basicAuthenticationIdentity = Thread.CurrentPrincipal.Identity as
BasicAuthenticationIdentity;

            if (basicAuthenticationIdentity != null)
            {
                basicAuthenticationIdentity.UserId = userId;

                return true;
            }
        }

        return false;
    }
}

```


Step 4: Basic Authentication on Controller

Since we already have our products controller,

```
public class ProductController : ApiController
{
    #region Private variable.

    private readonly IProductServices _productServices;

    #endregion

    #region Public Constructor

    /// <summary>
    /// Public constructor to initialize product service instance
    /// </summary>
    public ProductController(IProductServices productServices)
    {
        _productServices = productServices;
    }

    #endregion
```



```

// GET api/product

[GET("allproducts")]

[GET("all")]

public HttpResponseMessage Get()
{
    var products = _productServices.GetAllProducts();

    var productEntities = products as List<ProductEntity> ?? products.ToList();

    if (productEntities.Any())
        return Request.CreateResponse(HttpStatusCode.OK, productEntities);

    return Request.CreateErrorResponse(HttpStatusCode.NotFound, "Products not
found");
}

// GET api/product/5

[GET("productid/{id?}")]

[GET("particularproduct/{id?}")]

[GET("myproduct/{id:range(1, 3)}")]

public HttpResponseMessage Get(int id)
{
    var product = _productServices.GetProductById(id);

    if (product != null)
        return Request.CreateResponse(HttpStatusCode.OK, product);
}

```



```
        return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found  
for this id");
```

```
    }
```

```
// POST api/product
```

```
[POST("Create")]
```

```
[POST("Register")]
```

```
public int Post([FromBody] ProductEntity productEntity)
```

```
{
```

```
    return _productServices.CreateProduct(productEntity);
```

```
}
```

```
// PUT api/product/5
```

```
[PUT("Update/productid/{id}")]
```

```
[PUT("Modify/productid/{id}")]
```

```
public bool Put(int id, [FromBody] ProductEntity productEntity)
```

```
{
```

```
    if (id > 0)
```

```
    {
```

```
        return _productServices.UpdateProduct(id, productEntity);
```

```
    }
```

```
    return false;
```

```
}
```



```
// DELETE api/product/5

[DELETE("remove/productid/{id}")]

[DELETE("clear/productid/{id}")]

[PUT("delete/productid/{id}")]

public bool Delete(int id)
{
    if (id > 0)

        return _productServices.DeleteProduct(id);

    return false;
}
}
```

There are three ways in which you can use this authentication filter.

Just apply this filter to ProductController. You can add this filter at the top of the controller, for all API requests to be validated,

```
[ApiAuthenticationFilter]

[RoutePrefix("v1/Products/Product")]

public class ProductController : ApiController
```

You can also globally add this in Web API configuration file , so that filter applies to all the controllers and all the actions associated to it,


```
GlobalConfiguration.Configuration.Filters.Add(new ApiAuthenticationFilter());
```

You can also apply it to Action level too by your wish to apply or not apply authentication to that action,

```
// GET api/product

[ApiAuthenticationFilter(true)]

[GET("allproducts")]

[GET("all")]

public HttpResponseMessage Get()

{
    .....
}

// GET api/product/5

[ApiAuthenticationFilter(false)]

[GET("productid/{id?}")]

[GET("particularproduct/{id?}")]

[GET("myproduct/{id:range(1, 3)}")]

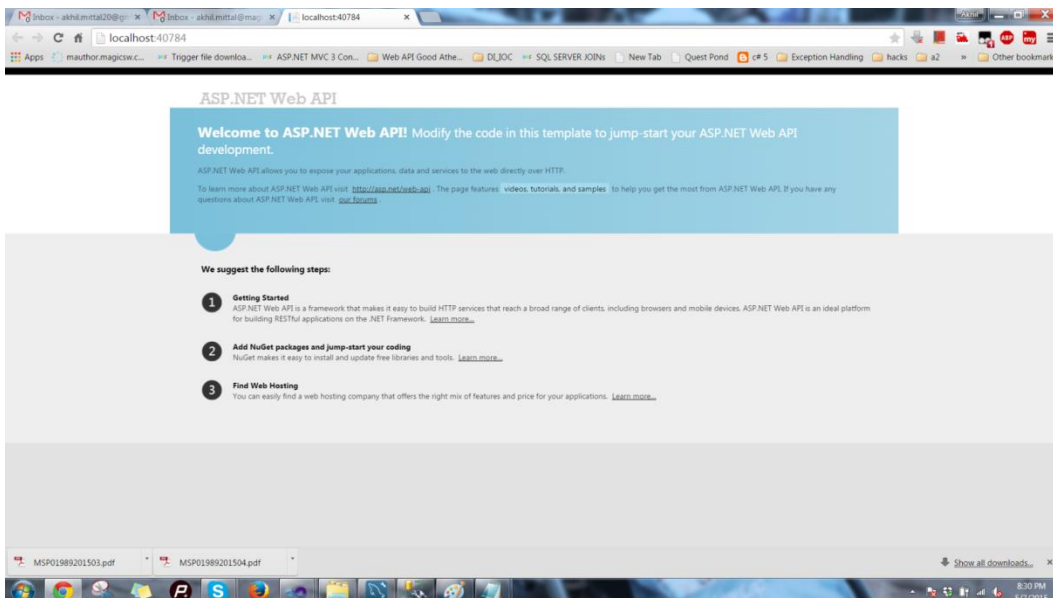
public HttpResponseMessage Get(int id)

{
    .....
}
```

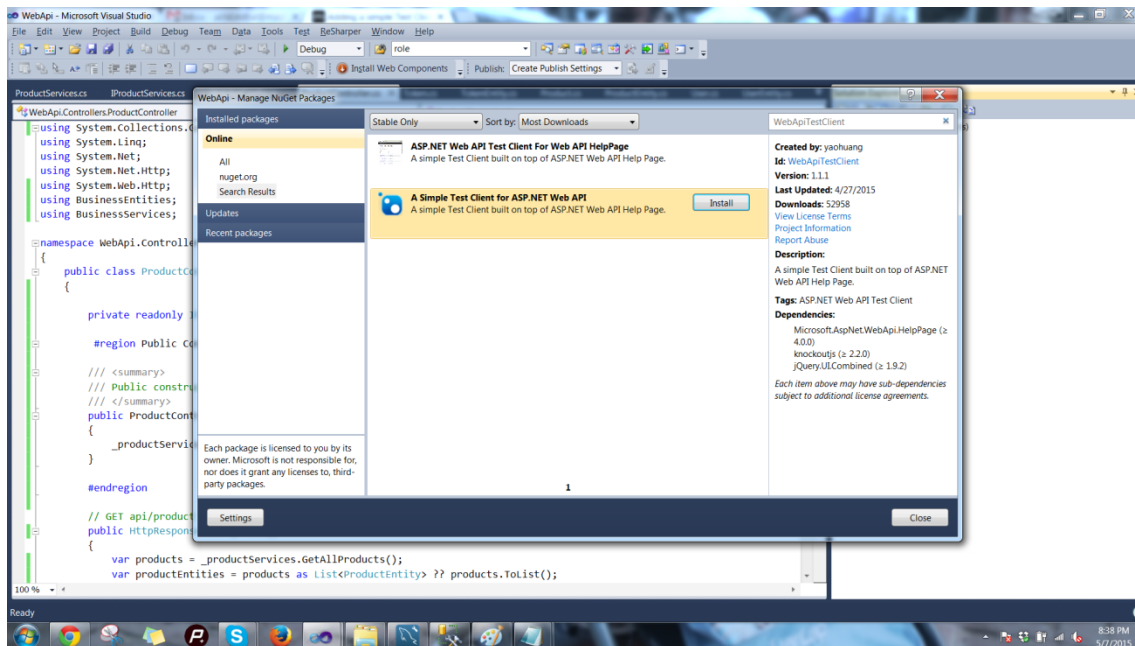

Running the application

We have already implemented Basic Authentication, just try to run the application to test if it is working

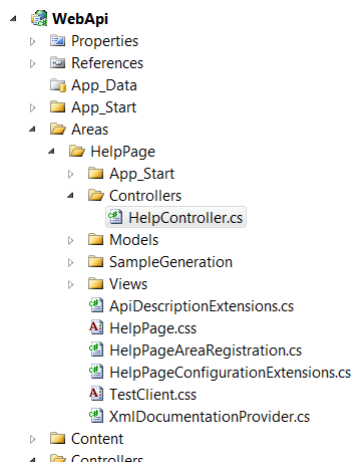
Just run the application, we get,



We already have our test client added, but for new readers, just go to Manage Nuget Packages, by right clicking WebAPI project and type WebAPITestClient in searchbox in online packages,



You'll get "A simple Test Client for ASP.NET Web API", just add it. You'll get a help controller in Areas-> HelpPage like shown below,



I have already provided the database scripts and data in my previous article, you can use the same.

Append "/help" in the application url, and you'll get the test client,

GET:

Product

API	Description
GET v1/Products/Product/all	Documentation for 'Get'.
GET v1/Products/Product/all?id={id}	Documentation for 'Get'.
GET v1/Products/Product/allproducts	Documentation for 'Get'.
GET v1/Products/Product/allproducts?id={id}	Documentation for 'Get'.
GET v1/Products/Product/myproduct/{id}	Documentation for 'Get'.
GET v1/Products/Product/particularproduct	Documentation for 'Get'.
GET v1/Products/Product/particularproduct/{id}	Documentation for 'Get'.
GET v1/Products/Product/productid	Documentation for 'Get'.
GET v1/Products/Product/productid/{id}	Documentation for 'Get'.

POST:

POST v1/Products/Product/Create	Documentation for 'Post'.
POST v1/Products/Product/Register	Documentation for 'Post'.

PUT:

[PUT v1/Products/Product/Update/productid/{id}](#)

Documentation for 'Put'.

[PUT v1/Products/Product/Modify/productid/{id}](#)

Documentation for 'Put'.

DELETE:[DELETE v1/Products/Product/delete/productid/{id}](#)

Documentation for 'Delete'.

[DELETE v1/Products/Product/remove/productid/{id}](#)

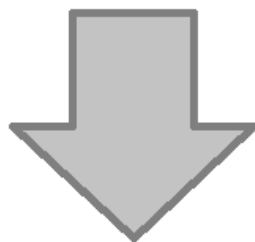
Documentation for 'Delete'.

[DELETE v1/Products/Product/clear/productid/{id}](#)

Documentation for 'Delete'.

You can test each service by clicking on it. Once you click on the service link, you'll be redirected to test the service page of that particular service. On that page there is a button Test API in the right bottom corner, just press that button to test your service,

**Press this button to
test the API**



A rectangular button with a light gray background and a thin black border. The text 'Test API' is centered on the button in a black, sans-serif font.

Service for Get All products,

[Help](#) [Page](#) [Home](#)

GET v1/Products/Product/all

Documentation for 'Get'.

GET v1/Products/Product/all ✕

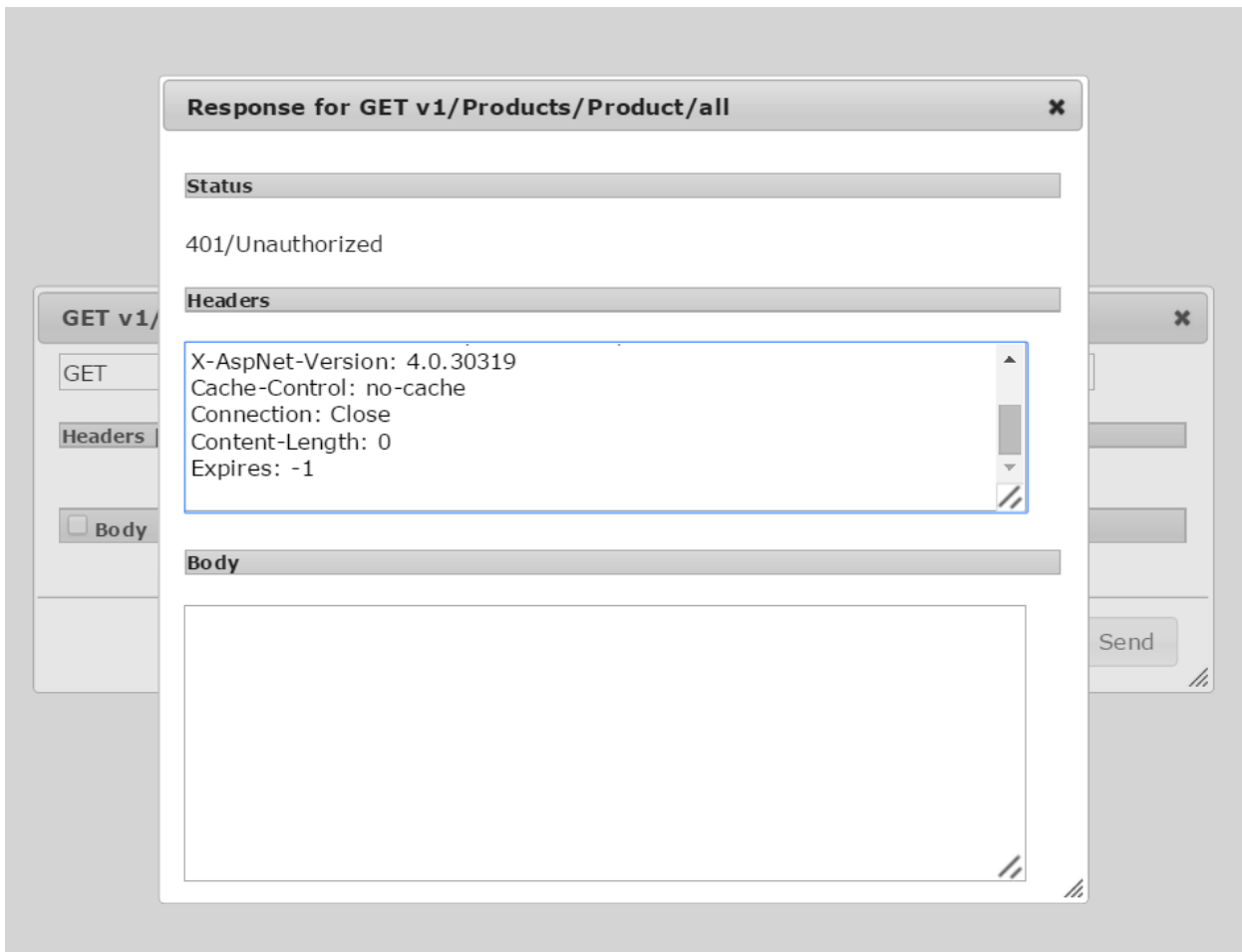
GET

Headers | [Add header](#)

☐ **Body**

Send

When you click on Send request, a popup will come asking Authentication required. Just cancel that popup and let request go without credentials. You'll get a response of Unauthorized i.e. 401,



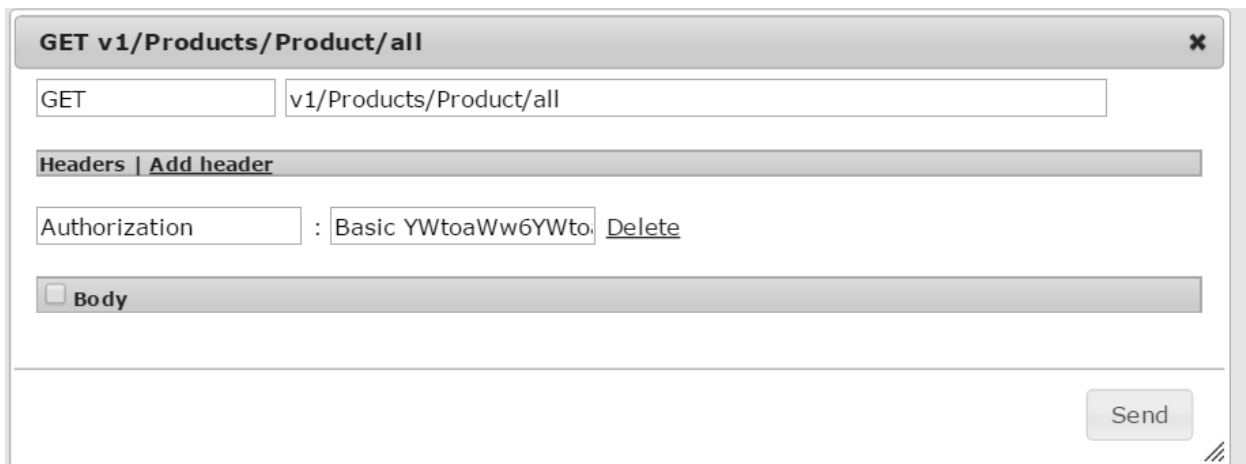
This means our authentication mechanism is working.



Just to double sure it, let's send the request with credentials now. Just add a header too with the request. Header should be like ,

Authorization : Basic YWtoaWw6YWtoaWw=

Here “YWtoaWw6YWtoaWw=” is my Base64 encoded user name and password i.e. akhil:akhil



The screenshot shows a REST client interface with a title bar "GET v1/Products/Product/all" and a close button. Below the title bar, there are two input fields: the first contains "GET" and the second contains "v1/Products/Product/all". Below these fields is a section labeled "Headers | Add header". Inside this section, there is a label "Authorization" followed by a colon and a text input field containing "Basic YWtoaWw6YWto". To the right of the input field is a "Delete" button. Below the headers section is a section labeled "Body" with a checkbox that is currently unchecked. At the bottom right of the interface is a "Send" button.

Click on Send and we get the response as desired,

Response for GET v1/Products/Product/all x

Status

200/OK

Headers

Server: ASP.NET Development Server/10.0.0.0
Date: Sun, 31 May 2015 09:11:10 GMT
X-AspNet-Version: 4.0.30319
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Content-Type: application/json; charset=utf-8

Body

```
{
  "ProductId": 1,
  "ProductName": "Laptop"
},
{
  "ProductId": 2,
  "ProductName": "computer"
},
{
  "ProductId": 4,
  "ProductName": "IPhone"
}
]
```



Likewise you can test all the service endpoints.

This means our service is working with Basic Authentication.

Design discrepancy

This design when running on SSL is very good for implementing Basic Authentication. But there are few scenarios in which, along with Basic Authentication I would like to leverage authorization too and not even authorization but sessions too. When we talk about creating an enterprise application, it just does not limit to securing our endpoint with authentication only.

In this design, each time I'll have to send user name and password with every request. Suppose I want to create such application, where authentication only occurs only once as my login is done and after successfully authenticated i.e. logging in I must be able to use other services of that application i.e. I am authorized now to use those services. Our application should be that robust that it restricts even authenticated user to use other services until he is not authorized. Yes I am talking about Token based authorization.

I'll expose only one endpoint for authentication and that will be my login service .So client only knows about that login service that needs credentials to get logged in to system.

After client successfully logs in I'll send a token that could be a GUID or an encrypted key by any xyz algorithm that I want when user makes request for any other services after login, should provide this token along with that request.

And to maintain sessions, our token will have an expiry too, that will last for 15 minutes, can be made configurable with the help of web.config file. After session is expired, user will be logged out, and will again have to use login service with credentials to get a new token. Seems exciting to me, let's implement this 😊

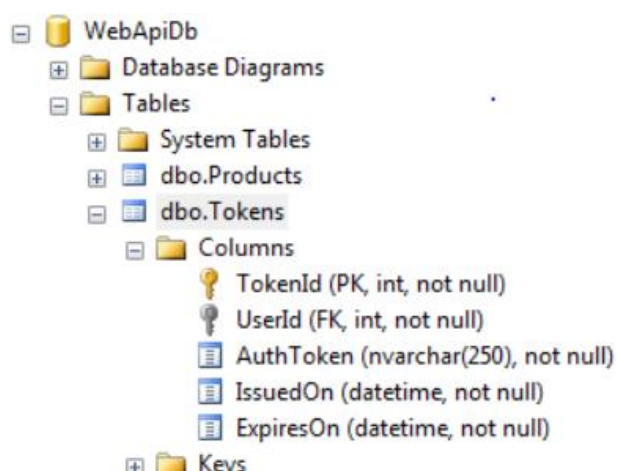
Implementing Token based Authorization

To overcome above mentioned scenarios, let's start developing and giving our application a shape of thick client enterprise architecture.



Set Database

Let's start with setting up a database. When we see our already created database that we had set up in [first](#) part of the series, we have a token table. We require this token table for token persistence. Our token will persist in database with an expiry time. If you are using your own database, you can create token table as,



Set Business Services

Just navigate to BusinessServices and create one more Interface named ITokenServices for token based operations,

```
using BusinessEntities;
```

```
namespace BusinessServices
```

```
{
```

```
    public interface ITokenServices
```

```
    {
```

```
        #region Interface member methods.
```

```
        /// <summary>
```

```
        /// Function to generate unique token with expiry against the provided userId.
```

```
        /// Also add a record in database for generated token.
```

```
        /// </summary>
```

```
        /// <param name="userId"></param>
```

```
        /// <returns></returns>
```

```
        TokenEntity GenerateToken(int userId);
```

```
        /// <summary>
```

```
        /// Function to validate token against expiry and existence in database.
```

```
        /// </summary>
```



```
    /// <param name="tokenId"></param>

    /// <returns></returns>

    bool ValidateToken(string tokenId);

    /// <summary>

    /// Method to kill the provided token id.

    /// </summary>

    /// <param name="tokenId"></param>

    bool Kill(string tokenId);

    /// <summary>

    /// Delete tokens for the specific deleted user

    /// </summary>

    /// <param name="userId"></param>

    /// <returns></returns>

    bool DeleteByUserId(int userId);

    #endregion

}

}
```

We have four methods defined in this interface. Let's create TokenServices class which implements ITokenServices and understand each method.

GenerateToken method takes userId as a parameter and generates a token, encapsulates that token in a token entity with Token expiry time and returns it to caller.

```
public TokenEntity GenerateToken(int userId)
{
    string token = Guid.NewGuid().ToString();

    DateTime issuedOn = DateTime.Now;

    DateTime expiredOn = DateTime.Now.AddSeconds(
Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));

    var tokendomain = new Token
    {
        UserId = userId,
        AuthToken = token,
        IssuedOn = issuedOn,
        ExpiresOn = expiredOn
    };

    _unitOfWork.TokenRepository.Insert(tokendomain);
    _unitOfWork.Save();

    var tokenModel = new TokenEntity()
    {
        UserId = userId,
        IssuedOn = issuedOn,
```



```

        ExpiresOn = expiredOn,

        AuthToken = token

    };

    return tokenModel;
}

```

While generating token, it names a database entry into Token table.

ValidateToken method just validates that the token associated with the request is valid or not i.e. it exists in the database within its expiry time limit.

```

public bool ValidateToken(string tokenId)
{
    var token = _unitOfWork.TokenRepository.Get(t => t.AuthToken == tokenId &&
t.ExpiresOn > DateTime.Now);

    if (token != null && !(DateTime.Now > token.ExpiresOn))
    {
        token.ExpiresOn = token.ExpiresOn.AddSeconds(
Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));

        _unitOfWork.TokenRepository.Update(token);

        _unitOfWork.Save();

        return true;
    }
}

```



```

    }

    return false;
}

```

It just takes token Id supplied in the request.

Kill Token just kills the token i.e. removes the token from database.

```

public bool Kill(string tokenId)
{
    _unitOfWork.TokenRepository.Delete(x => x.AuthToken == tokenId);
    _unitOfWork.Save();

    var isNotDeleted = _unitOfWork.TokenRepository.GetMany(x => x.AuthToken ==
tokenId).Any();

    if (isNotDeleted) { return false; }

    return true;
}

```

DeleteByUserId method deletes all token entries from the database w.r.t particular userId associated with those tokens.

```

public bool DeleteByUserId(int userId)
{

```



```

_unitOfWork.TokenRepository.Delete(x => x.UserId == userId);

_unitOfWork.Save();

```

```

var isNotDeleted = _unitOfWork.TokenRepository.GetMany(x => x.UserId == userId).Any();

return !isNotDeleted;

}

```

So with `_unitOfWork` and along with Constructor our class becomes,

```

using System;

using System.Configuration;

using System.Linq;

using BusinessEntities;

using DataModel;

using DataModel.UnitOfWork;

namespace BusinessServices
{

public class TokenServices:ITokenServices
{

#region Private member variables.

private readonly UnitOfWork _unitOfWork;

#endregionregion

```


#region Public constructor.

/// <summary>

/// Public constructor.

/// </summary>

public TokenServices(**UnitOfWork** unitOfWork)

{

_unitOfWork = unitOfWork;

}

#endregion

#region Public member methods.

/// <summary>

/// Function to generate unique token with expiry against the provided userId.

/// Also add a record in database for generated token.

/// </summary>

/// <param name="userId"></param>

/// <returns></returns>

public **TokenEntity** GenerateToken(**int** userId)

{

string token = **Guid**.NewGuid().ToString();


```
DateTime issuedOn = DateTime.Now;

DateTime expiredOn = DateTime.Now.AddSeconds(
    Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));

var tokendomain = new Token
{
    UserId = userId,
    AuthToken = token,
    IssuedOn = issuedOn,
    ExpiresOn = expiredOn
};

_unitOfWork.TokenRepository.Insert(tokendomain);
_unitOfWork.Save();

var tokenModel = new TokenEntity()
{
    UserId = userId,
    IssuedOn = issuedOn,
    ExpiresOn = expiredOn,
    AuthToken = token
};

return tokenModel;
}
```



```

/// <summary>

/// Method to validate token against expiry and existence in database.

/// </summary>

/// <param name="tokenId"></param>

/// <returns></returns>

public bool ValidateToken(string tokenId)

{

var token = _unitOfWork.TokenRepository.Get(t => t.AuthToken == tokenId && t.ExpiresOn
> DateTime.Now);

if (token != null && !(DateTime.Now > token.ExpiresOn))

{

token.ExpiresOn = token.ExpiresOn.AddSeconds(

Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));

_unitOfWork.TokenRepository.Update(token);

_unitOfWork.Save();

return true;

}

return false;

}

/// <summary>

/// Method to kill the provided token id.

```



```

/// </summary>

/// <param name="tokenId">true for successful delete</param>

public bool Kill(string tokenId)
{
    _unitOfWork.TokenRepository.Delete(x => x.AuthToken == tokenId);

    _unitOfWork.Save();

    var isNotDeleted = _unitOfWork.TokenRepository.GetMany(x => x.AuthToken ==
tokenId).Any();

    if (isNotDeleted) { return false; }

    return true;
}

```

```

/// <summary>

/// Delete tokens for the specific deleted user

/// </summary>

/// <param name="userId"></param>

/// <returns>true for successful delete</returns>

public bool DeleteByUserId(int userId)
{
    _unitOfWork.TokenRepository.Delete(x => x.UserId == userId);

    _unitOfWork.Save();

    var isNotDeleted = _unitOfWork.TokenRepository.GetMany(x => x.UserId == userId).Any();

```



```
return !isNotDeleted;

}
```

```
#endregion

}

}
```

Do not forget to resolve the dependency of this Token service in DependencyResolver class. Add `registerComponent.RegisterType<ITokenServices, TokenServices>();` to the `SetUp` method of `DependencyResolver` class in `BusinessServices` project.

```
[Export(typeof(IComponent))]

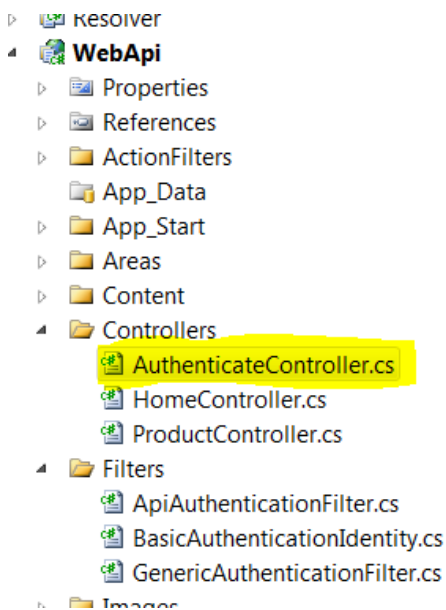
public class DependencyResolver : IComponent
{
    public void SetUp(IRegisterComponent registerComponent)
    {
        registerComponent.RegisterType<IProductServices, ProductServices>();
        registerComponent.RegisterType<IUserServices, UserServices>();
        registerComponent.RegisterType<ITokenServices, TokenServices>();

    }
}
```


Setup WebAPI/Controller:

Now since we decided, that we don't want authentication to be applied on each and every API exposed, I'll create a single Controller/API endpoint that takes authentication or login request and makes use of Token Service to generate token and respond client/caller with a token that persists in database with expiry details.

Add a new Controller under Controllers folder in WebAPI with a name Authenticate,



AuthenticateController:

```
using System.Configuration;
```

```
using System.Net;
```

```
using System.Net.Http;
```

```
using System.Web.Http;
```

```
using AttributeRouting.Web.Http;
```

```
using BusinessServices;
```



```
using WebApi.Filters;
```

```
namespace WebApi.Controllers
```

```
{
```

```
    [ApiAuthenticationFilter]
```

```
    public class AuthenticateController : ApiController
```

```
    {
```

```
        #region Private variable.
```

```
        private readonly ITokenServices _tokenServices;
```

```
    #endregion
```

```
    #region Public Constructor
```

```
        /// <summary>
```

```
        /// Public constructor to initialize product service instance
```

```
        /// </summary>
```

```
        public AuthenticateController(ITokenServices tokenServices)
```

```
        {
```

```
            _tokenServices = tokenServices;
```

```
        }
```



```
#endregion
```

```
/// <summary>
```

```
/// Authenticates user and returns token with expiry.
```

```
/// </summary>
```

```
/// <returns></returns>
```

```
[POST("login")]
```

```
[POST("authenticate")]
```

```
[POST("get/token")]
```

```
public HttpResponseMessage Authenticate()
```

```
{
```

```
    if (System.Threading.Thread.CurrentPrincipal!=null &&
        System.Threading.Thread.CurrentPrincipal.Identity.IsAuthenticated)
```

```
    {
```

```
        var basicAuthenticationIdentity =
        System.Threading.Thread.CurrentPrincipal.Identity as BasicAuthenticationIdentity;
```

```
        if (basicAuthenticationIdentity != null)
```

```
        {
```

```
            var userId = basicAuthenticationIdentity.UserId;
```

```
            return GetAuthToken(userId);
```

```
        }
```

```
    }
```

```
    return null;
```



```

    }

    /// <summary>
    /// Returns auth token for the validated user.
    /// </summary>
    /// <param name="userId"></param>
    /// <returns></returns>
    private HttpResponseMessage GetAuthToken(int userId)
    {
        var token = _tokenServices.GenerateToken(userId);
        var response = Request.CreateResponse(HttpStatusCode.OK, "Authorized");
        response.Headers.Add("Token", token.AuthToken);
        response.Headers.Add("TokenExpiry",
ConfigurationManager.AppSettings["AuthTokenExpiry"]);
        response.Headers.Add("Access-Control-Expose-Headers", "Token,TokenExpiry" );
        return response;
    }
}
}
}

```

The controller is decorated with our authentication filter,

```
[ApiAuthenticationFilter]
```



```
public class AuthenticateController : ApiController
```

So, each and every request coming through this controller will have to pass through this authentication filter, that check for BasicAuthentication header and credentials. Authentication filter sets CurrentThread principal to the authenticated Identity.

There is a single Authenticate method / action in this controller. You can decorate it with multiple endpoints like we discussed in [fourth](#) part of the series,

```
[POST("login")]
```

```
[POST("authenticate")]
```

```
[POST("get/token")]
```

Authenticate method first checks for CurrentThreadPrincipal and if the user is authenticated or not i.e job done by authentication filter,

```
if (System.Threading.Thread.CurrentPrincipal != null &&  
    System.Threading.Thread.CurrentPrincipal.Identity.IsAuthenticated)
```

When it finds that the user is authenticated, it generates an auth token with the help of TokenServices and returns user with Token and its expiry,

```
response.Headers.Add("Token", token.AuthToken);
```

```
response.Headers.Add("TokenExpiry",  
    ConfigurationManager.AppSettings["AuthTokenExpiry"]);
```



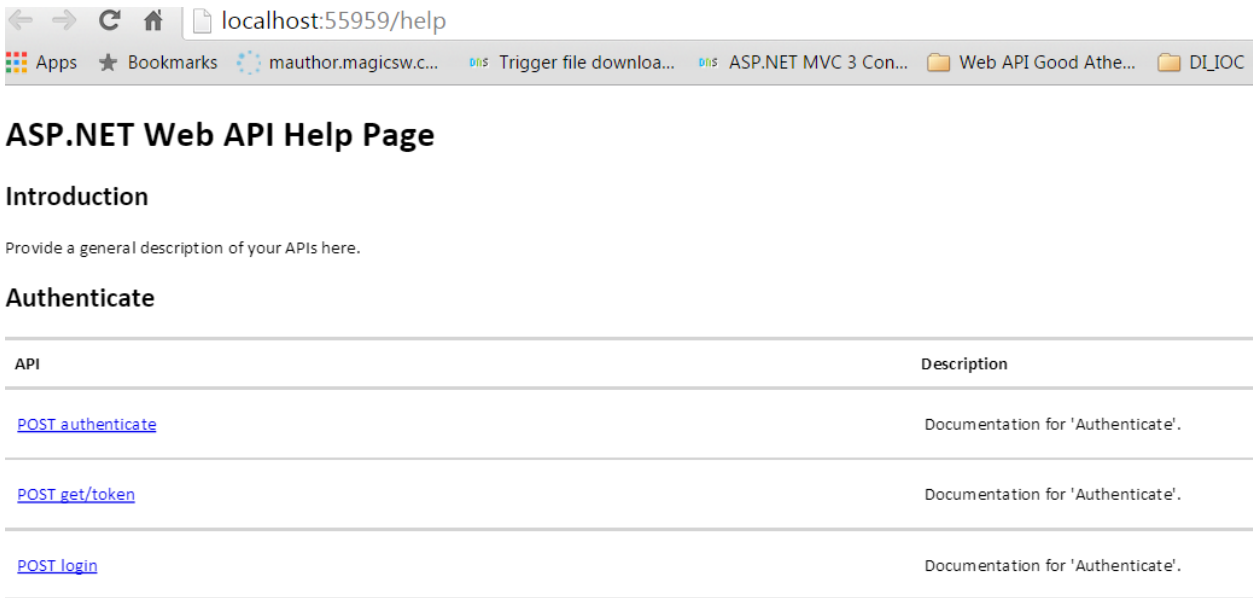
```
response.Headers.Add("Access-Control-Expose-Headers", "Token,TokenExpiry" );  
  
return response;
```

In our BasicAuthenticationIdentity class, I purposely used userId property so that we can make use of this property when we try to generate token, that we are doing in this controller's Authenticate method,

```
var basicAuthenticationIdentity = System.Threading.Thread.CurrentPrincipal.Identity as  
BasicAuthenticationIdentity;  
  
if (basicAuthenticationIdentity != null)  
{  
  
var userId = basicAuthenticationIdentity.UserId;  
  
return GetAuthToken(userId);  
  
}
```

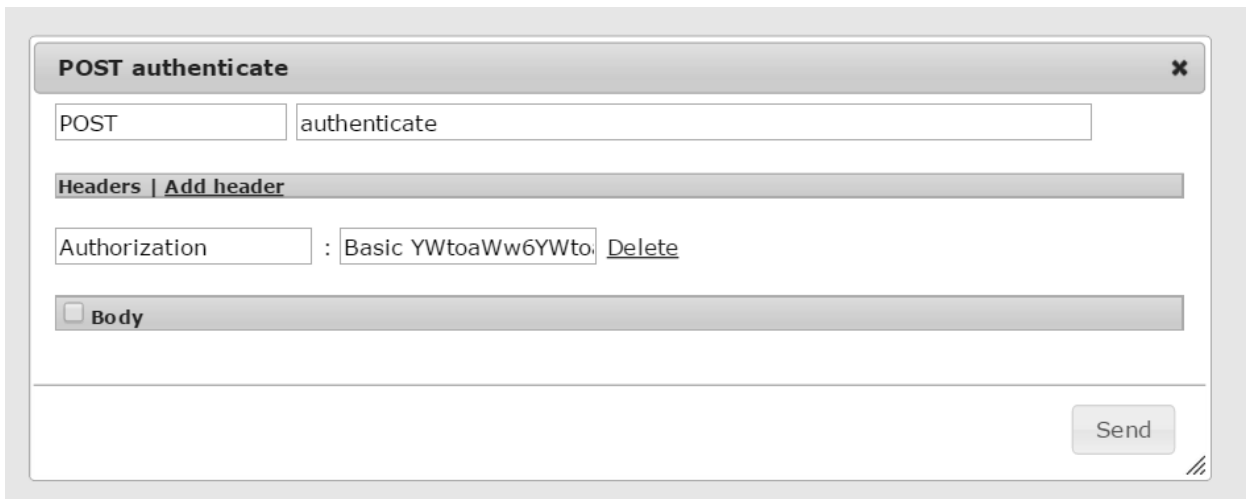
Now when you run this application, You'll see Authenticate api as well, just invoke this api with Basic Authentication and User credentials, you'll get the token with expiry, let's do this step by step.

1. Run the application.



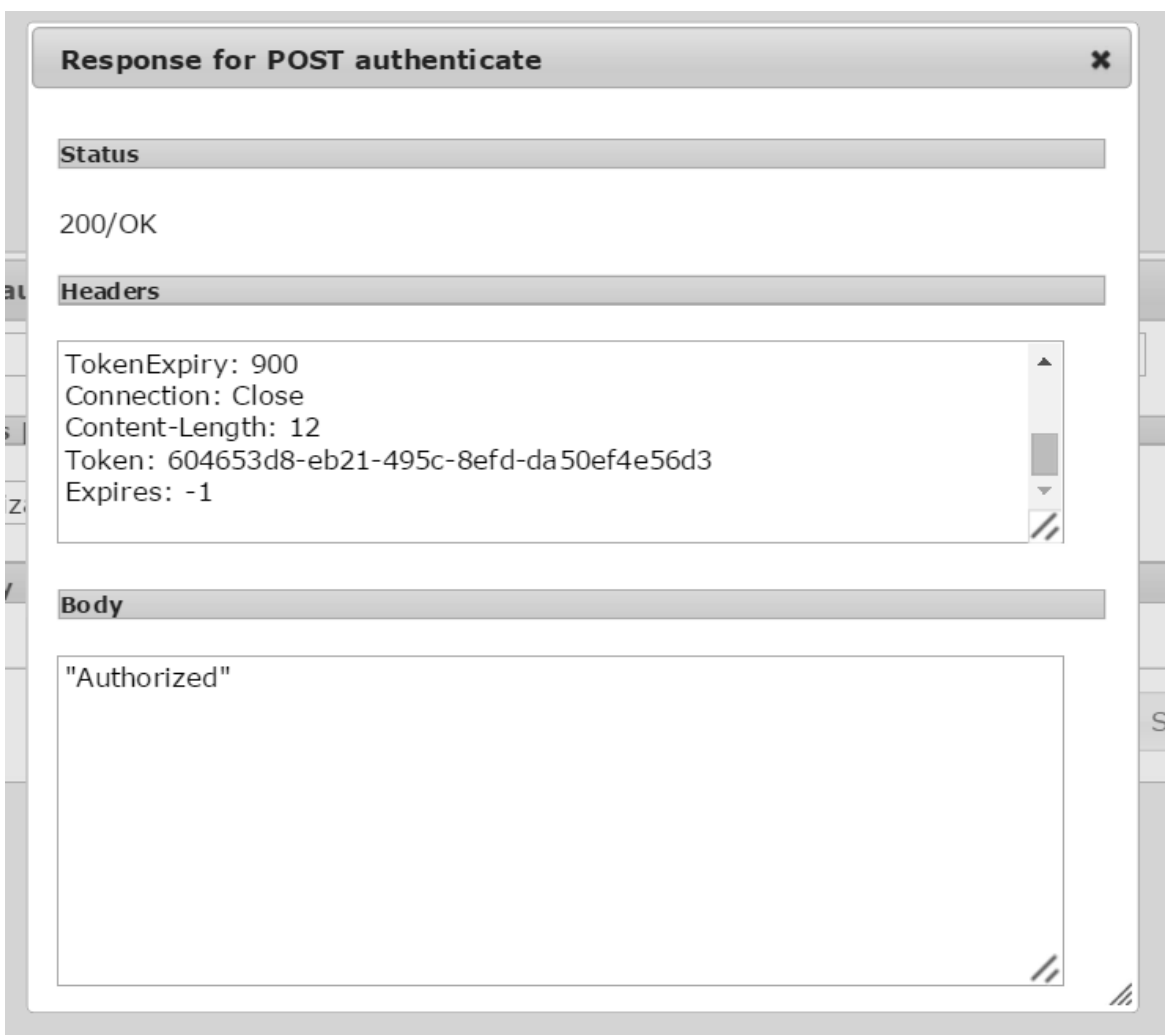
2. Click on first api link i.e. [POST authenticate](#). You'll get the page to test the api,

3. Press the TestAPI button in the right corner. In the test console, provide Header information with Authorization as Basic and user credentials in Base64 format, like we did earlier. Click on Send.



The screenshot shows a REST client window titled "POST authenticate". It has a "POST" method selected and the URL "authenticate". Below the URL bar is a "Headers" section with an "Add header" button. A header is added: "Authorization : Basic YWtoaWw6YWto" with a "Delete" button next to it. There is a "Body" section with a checkbox and the label "Body". A "Send" button is at the bottom right.

4. Now since we have provided valid credentials ,we'll get a token from the Authenticate controller, with its expiry time,



The screenshot shows a REST client window titled "Response for POST authenticate". It displays the response details for the POST request. The "Status" is "200/OK". The "Headers" section shows: "TokenExpiry: 900", "Connection: Close", "Content-Length: 12", "Token: 604653d8-eb21-495c-8efd-da50ef4e56d3", and "Expires: -1". The "Body" section shows the response text: "Authorized".

In database,

69B5ZR1.WebApiDb - dbo.Tokens		SQLQuery1.sql - 6...aster (akhil (53))			
	TokenId	UserId	AuthToken	IssuedOn	ExpiresOn
►	1	1	604653d8-eb21-495c-8efd-da50ef4e56d3	2015-06-23 13:54:21.520	2015-06-23 14:09:21.520
*	NULL	NULL	NULL	NULL	NULL

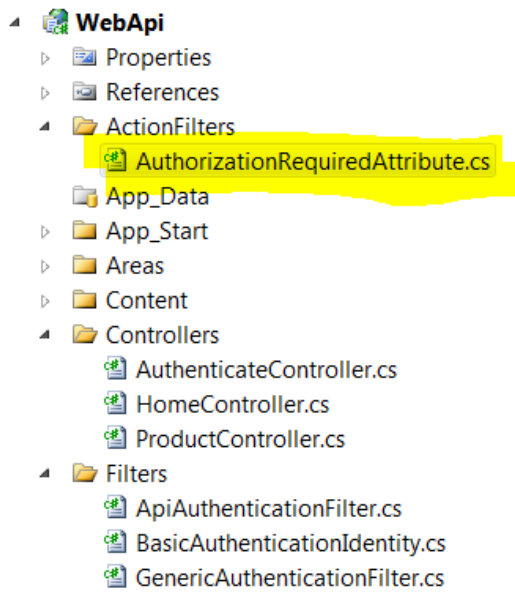
Here we get response 200, i.e. our user is authenticated and logged into system. TokenExpiry in 900 i.e. 15 minutes. Note that the time difference between IssuedOn and ExpiresOn is 15 minutes, this we did in TokenServices class method GenerateToken, you can set the time as per your need. Token is 604653d8-eb21-495c-8efd-da50ef4e56d3. Now for 15 minutes we can use this token to call our other services. But before that we should mark our other services to understand this token and respond accordingly. Keep the generated token saved so that we can use it further in calling other services that I am about to explain. So let's setup authorization on other services.

Setup Authorization Action Filter

We already have our Authentication filter in place and we don't want to use it for authorization purposes. So we have to create a new Action Filter for authorization. This action filter will only recognize Token coming in requests. It assumes that, requests are already authenticated through our login channel, and now user is authorized/not authorized to use other services like Products in our case, there could be a number of other services too, which can use this authorization action filter. For request to get authorized, now we don't have to pass user credentials. Only token (received from Authenticate controller after successful validation) needs to be passed through request.

Add a folder named ActionFilters in WebAPI project. And add a class named [AuthorizationRequiredAttribute](#)

Deriving from ActionFilterAttribute,



Override the OnActionExecuting method of ActionFilterAttribute, this is the way we define an action filter in API project.

```
using System.Linq;
```

```
using System.Net;
```

```
using System.Net.Http;
```

```
using System.Web.Http.Controllers;
```

```
using System.Web.Http.Filters;
```

```
using BusinessServices;
```

```
namespace WebApi.ActionFilters
```

```
{
```



```
public class AuthorizationRequiredAttribute : ActionFilterAttribute
{
    private const string Token = "Token";

    public override void OnActionExecuting(HttpContext filterContext)
    {
        // Get API key provider
        var provider = filterContext.ControllerContext.Configuration
            .DependencyResolver.GetService(typeof(ITokenServices)) as ITokenServices;

        if (filterContext.Request.Headers.Contains(Token))
        {
            var tokenValue = filterContext.Request.Headers.GetValues(Token).First();

            // Validate Token
            if (provider != null && !provider.ValidateToken(tokenValue))
            {
                var responseMessage = new HttpResponseMessage(HttpStatusCode.Unauthorized) {
                    ReasonPhrase = "Invalid Request" };
                filterContext.Response = responseMessage;
            }
        }
        else
```



```

{

filterContext.Response = new HttpResponseMessage(HttpStatusCode.Unauthorized);

}

base.OnActionExecuting(filterContext);

}

}

}

```

The overridden method checks for “Token” attribute in the Header of every request, if token is present, it calls ValidateToken method from TokenServices to check if the token exists in the database. If token is valid, our request is navigated to the actual controller and action that we requested, else you’ll get an error message saying unauthorized.

Mark Controllers with Authorization filter

We have our action filter ready. Now let’s mark our controller ProductController with this attribute. Just open Product controller class and at the top just decorate that class with this ActionFilter attribute,

```

[AuthorizationRequired]

[RoutePrefix("v1/Products/Product")]

public class ProductController : ApiController

```



```
{
```

We have marked our controller with the action filter that we created, now every request coming to the actions of this controller will have to be passed through this ActionFilter, that checks for the token in request.

You can mark other controllers as well with the same attribute, or you can do marking at action level as well. Suppose you want certain actions should be available to all users irrespective of their authorization then you can just mark only those actions which require authorization and leave other actions as they are like I explained in Step 4 of Implementing Basic Authentication.

Maintaining Session using Token

We can certainly use these tokens to maintain session as well. The tokens are issued for 900 seconds i.e. 15 minutes. Now we want that user should continue to use this token if he is using other services as well for our application. Or suppose there is a case where we only want user to finish his work on the site within 15 minutes or within his session time before he makes a new request. So while validating token in TokenServices, what I have done is, to increase the time of the token by 900 seconds whenever a valid request comes with a valid token,

```
/// <summary>
/// Method to validate token against expiry and existence in database.
/// </summary>
/// <param name="tokenId"></param>
/// <returns></returns>
public bool ValidateToken(string tokenId)
{
```



```
var token = _unitOfWork.TokenRepository.Get(t => t.AuthToken == tokenId &&
t.ExpiresOn > DateTime.Now);
```

```
if (token != null && !(DateTime.Now > token.ExpiresOn))
```

```
{
```

```
    token.ExpiresOn = token.ExpiresOn.AddSeconds(
```

```
Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));
```

```
    _unitOfWork.TokenRepository.Update(token);
```

```
    _unitOfWork.Save();
```

```
    return true;
```

```
}
```

```
return false;
```

```
}
```

In above code for token validation, first we check if the requested token exists in the database and is not expired. We check expiry by comparing it with current date time. If it is valid token we just update the token into database with a new ExpiresOn time that is adding 900 seconds.

```
if (token != null && !(DateTime.Now > token.ExpiresOn))
```

```
{
```

```
    token.ExpiresOn = token.ExpiresOn.AddSeconds(
```

```
Convert.ToDouble(ConfigurationManager.AppSettings["AuthTokenExpiry"]));
```

```
    _unitOfWork.TokenRepository.Update(token);
```



```
_unitOfWork.Save();
```

By doing this we can allow end user or client to maintain session and keep using our services/application with a session timeout of 15 minutes. This approach can also be leveraged in multiple ways, like making different services with different session timeouts or many such conditions could be applied when we work on real time application using APIs.

Running the application

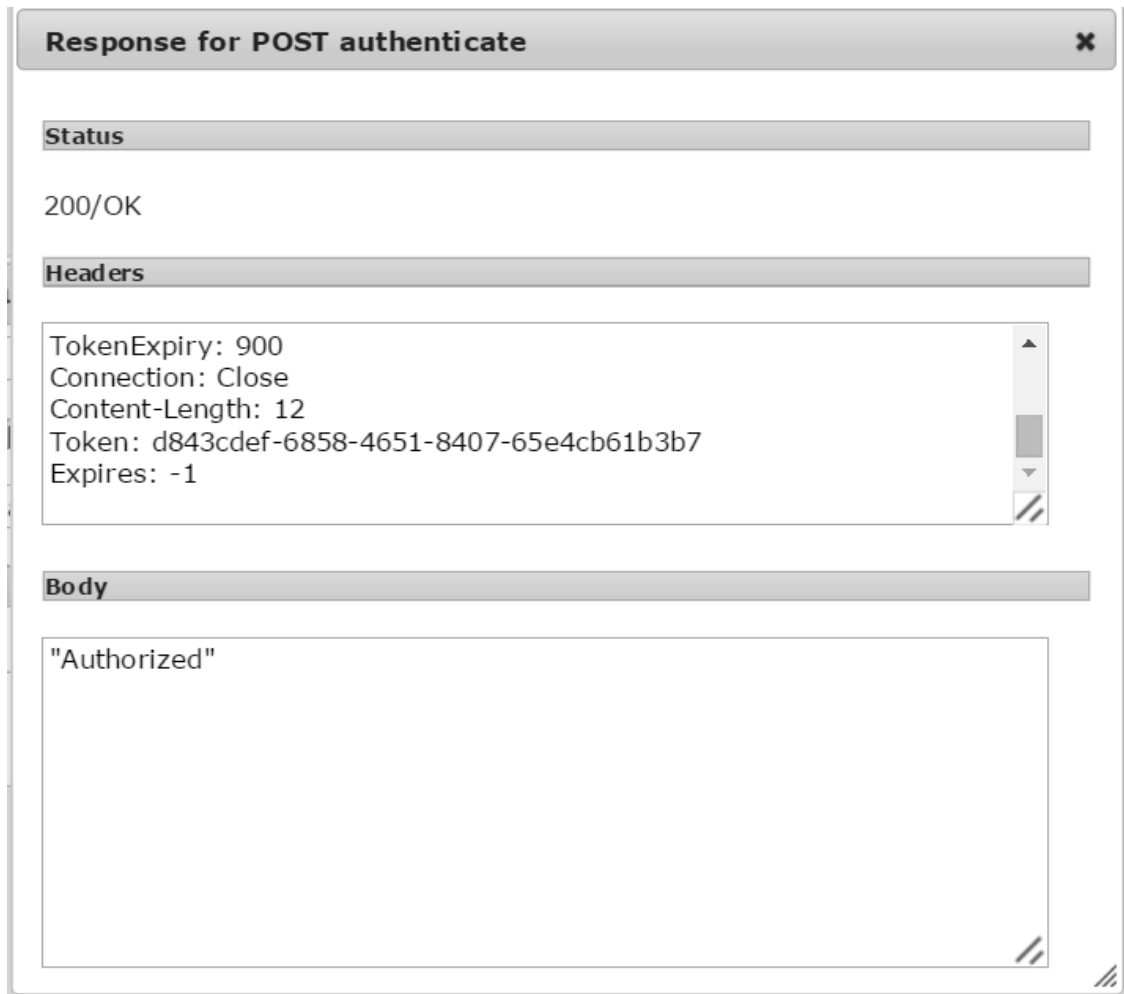
Our job is almost done.



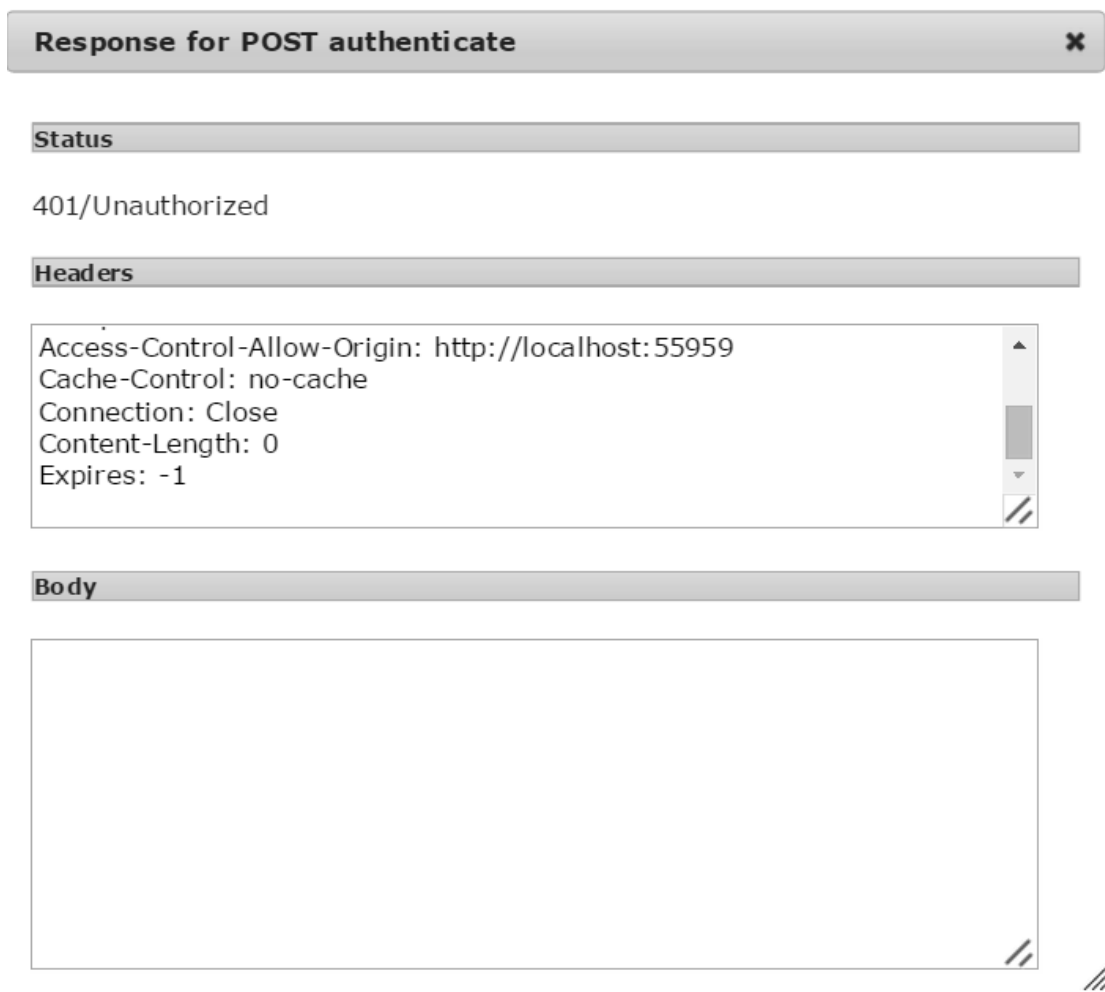
We just need to run the application and test if it is working fine or not. If you have saved the token you generated earlier while testing authentication you can use same to test authorization. I am just again running the whole cycle to test application.

Test Authentication

Repeat the tests we did earlier to get Auth Token. Just invoke the Authenticate controller with valid credentials and Basic authorization header. I get,



And without providing Authorization header as basic with credentials I get,



I just saved the token that I got in first request.

Now try to call ProductController actions.

Test Authorization

Run the application to invoke Product Controller actions. Try to invoke them without providing any Token,

Product

API	Description
GET v1/Products/Product/all	Documentation for 'Get'.
GET v1/Products/Product/all?id={id}	Documentation for 'Get'.
GET v1/Products/Product/allproducts	Documentation for 'Get'.
GET v1/Products/Product/allproducts?id={id}	Documentation for 'Get'.

Invoke first service in the list,

GET v1/Products/Product/all

GET

v1/Products/Product/all

Headers

Add header

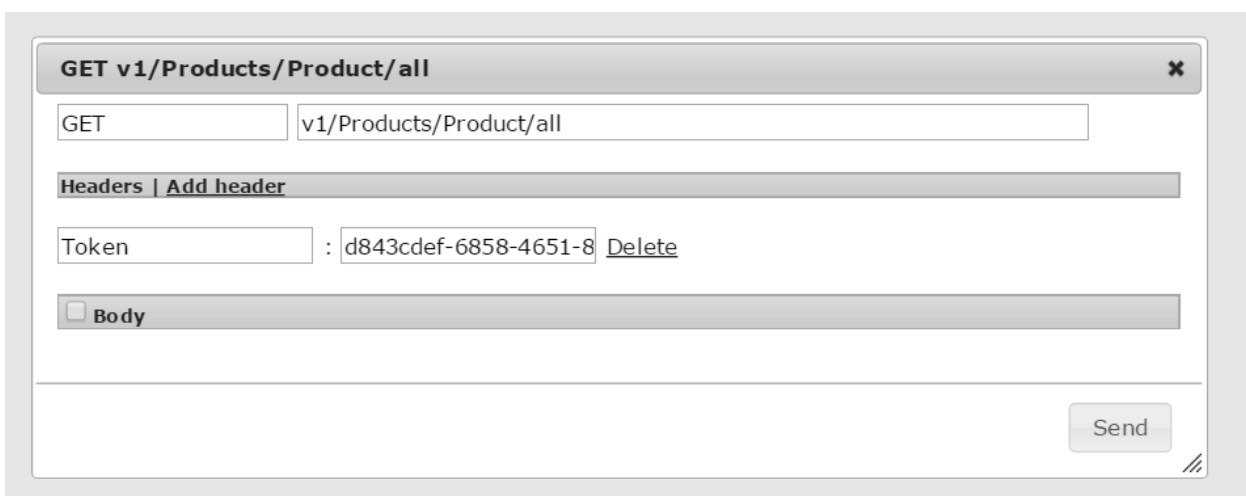
☐ Body

Send

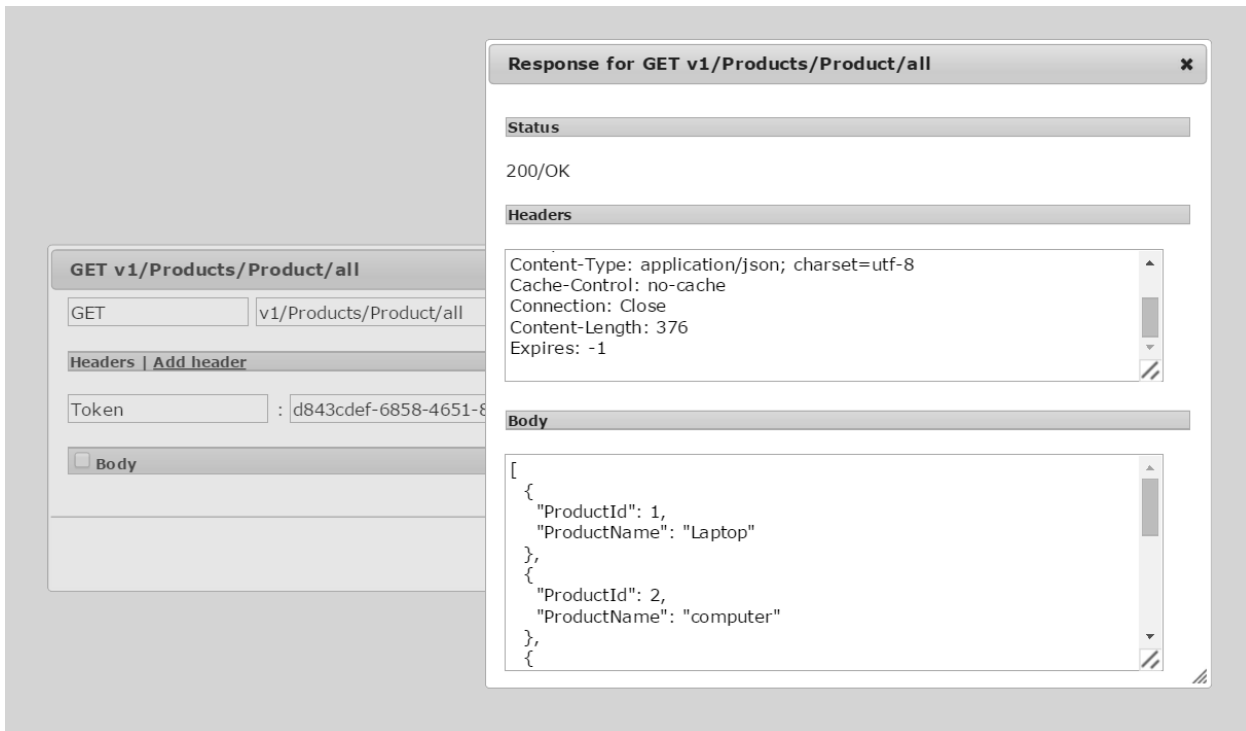
Click send,



Here we get Unauthorized, i.e. because our ProductController is marked with authorization attribute that checks for a Token. So here our request is invalid. Now try calling this action by providing the token that we saved,



Click on Send and we get,



That means we got the response and our token was valid. Now we see our Authentication and Authorization, both functionalities are working fine. You can test Sessions by your own.



Likewise you can test all actions. You can create other controllers and test the security, and play around with different set of permutations and combinations.

Conclusion

We covered and learnt a lot. In this article I tried to explain about how we can build an API application with basic Authentication and Authorization. One can mould this concept to achieve the level of security needed. Like token generation mechanism could be customized as per one's requirement. Two level of security could be implemented where authentication and authorization is needed for every service. One can also implement authorization on actions based on Roles.



Image credit: http://www.greencountryfordofparsons.com/images/secure_application.jpg

I already stated that there is no specific way of achieving security, the more you understand the concept, the more secure system you can make. The techniques used in this article or the design implemented in this article should be leveraged well if you use it with SSL (Secure Socket Layer), running the REST apis on https. In next chapter I'll try to explain some more beautiful implementations and concepts. Till then Happy Coding 😊

You can also download the complete source code with all packages from [Github](#).

References

<https://msdn.microsoft.com/en-us/magazine/dn781361.aspx>

<http://weblog.west-wind.com/posts/2013/Apr/18/A-WebAPI-Basic-Authentication-Authorization-Filter>

Request Logging and Exception Handling/Logging in Web APIs Using Action Filters, Exception Filters and NLog

This article of the book will explain how we can handle requests and log them for tracking and for the sake of debugging, how we can handle exceptions and log them. We'll follow centralized way of handling exceptions in WebAPI and write our custom classes to be mapped to the type of exception that we encounter and log the accordingly. I'll use [NLog](#) to log requests and exceptions as well. We'll leverage the capabilities of Exception Filters and Action Filters to centralize request logging and exception handling in WebAPI.

Request Logging

Since we are writing web services, we are exposing our end points. We must know where the requests are coming from and what requests are coming to our server. Logging could be very beneficial and helps us in a lot of ways like, debugging, tracing, monitoring and analytics.

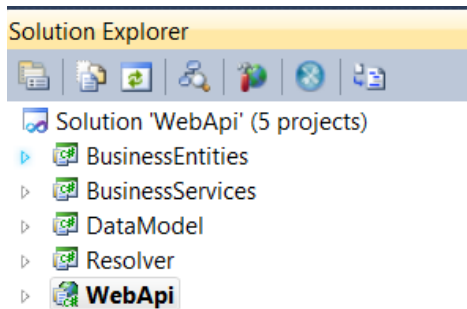


We already have an existing design. If you open the solution, you'll get to see the structure as mentioned below or one can also implement this approach in their existing solution as well,

Setup NLog in WebAPI

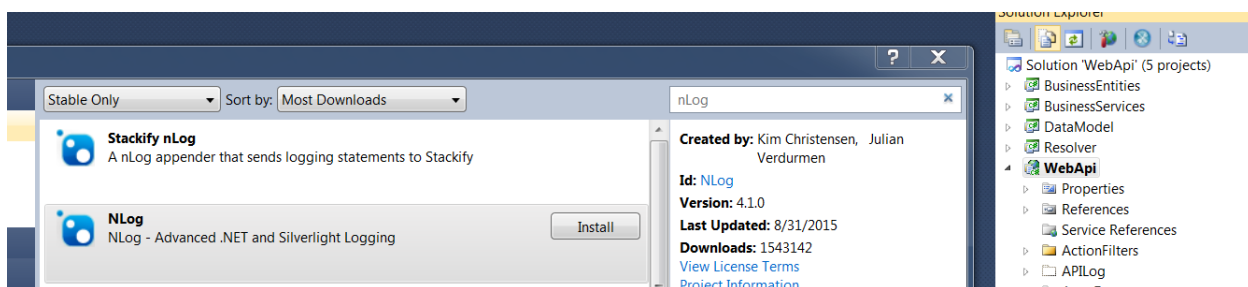
NLog serves various purposes but primarily logging. We'll use NLog for logging into files and windows event as well. You can read more about NLog at <http://NLog-project.org/>

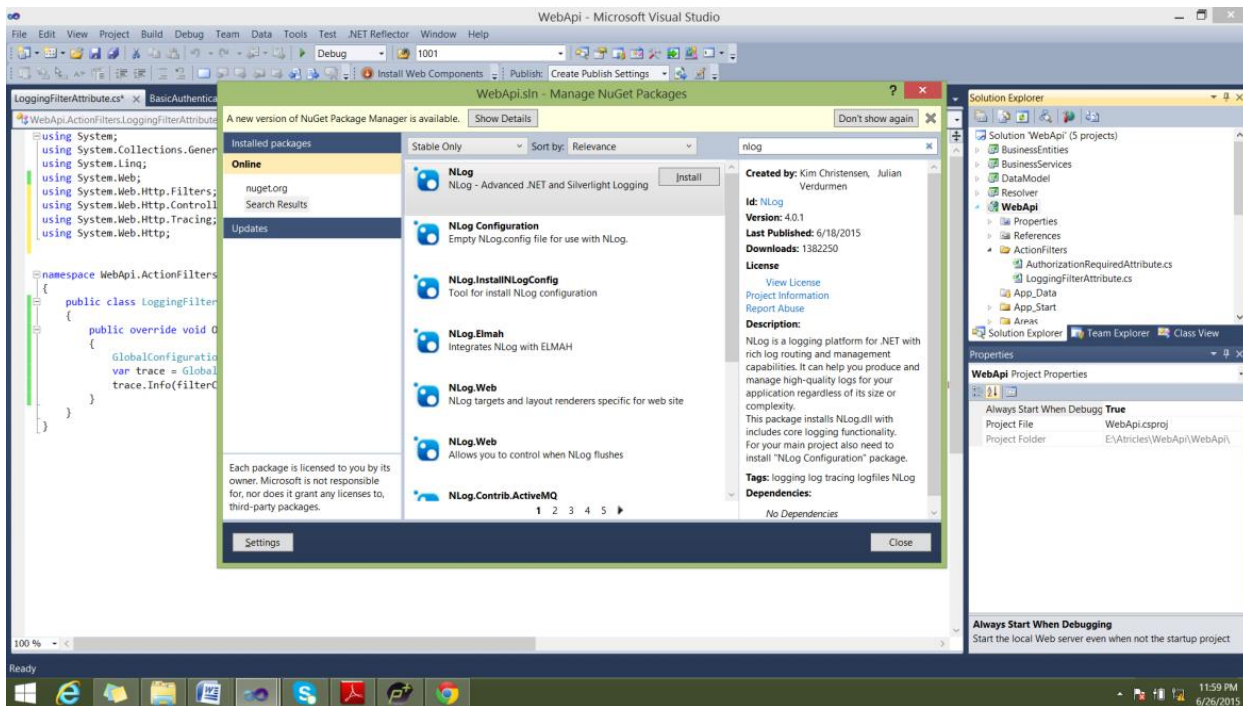
One can use the sample application that we used in [Day#5](#) or can have any other application as well. I am using the existing sample application that we were following throughout all the parts of this series. Our application structure looks something like,



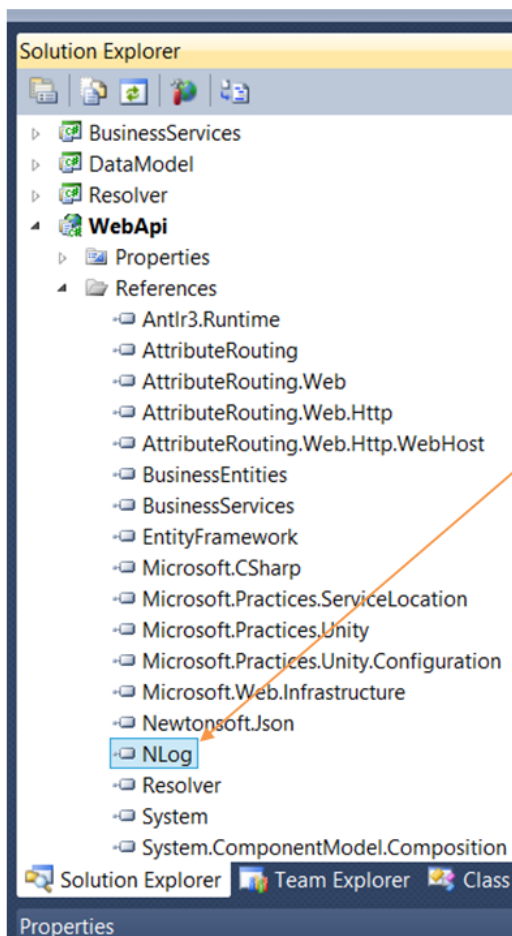
Step 1: Download NLog Package

Right click WebAPI project and select manage Nuget Packages from the list. When the Nuget Package Manager appears, search for NLog. You'll get NLog like shown in image below, just install it to our project,






After adding this you will find following NLog dll referenced in your application –



Step 2: Configuring NLog

To configure NLog with application add following settings in our existing WebAPI web.config file,

ConfigSection –



```
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework" />
    <section name="nlog" type="NLog.Config.ConfigSectionHandler, NLog" />
  </configSections>
```

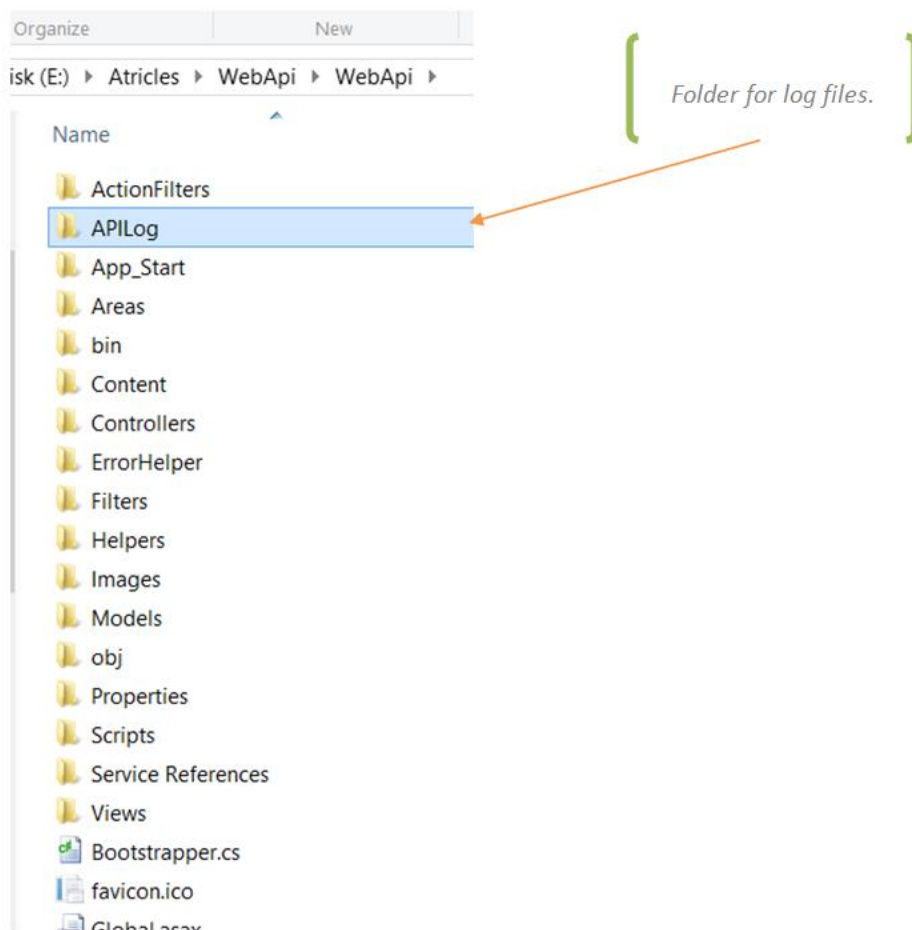
Configuration Section – I have added the <NLog> section to configuration and defined the path and format dynamic target log file name, also added the eventlog source to Api Services.

Dynamic log file name.



```
<nlog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <targets>
    <target name="logfile" xsi:type="File" fileName="${basedir}/APILog/${date:format=yyyy-MM-dd}-api.log" />
    <target name="eventlog" xsi:type="EventLog" layout="${message}" log="Application" source="Api Services" />
  </targets>
  <rules>
    <logger name="*" minlevel="Trace" writeTo="logfile" />
    <logger name="*" minlevel="Trace" writeTo="eventlog" />
  </rules>
</nlog>
</configuration>
```

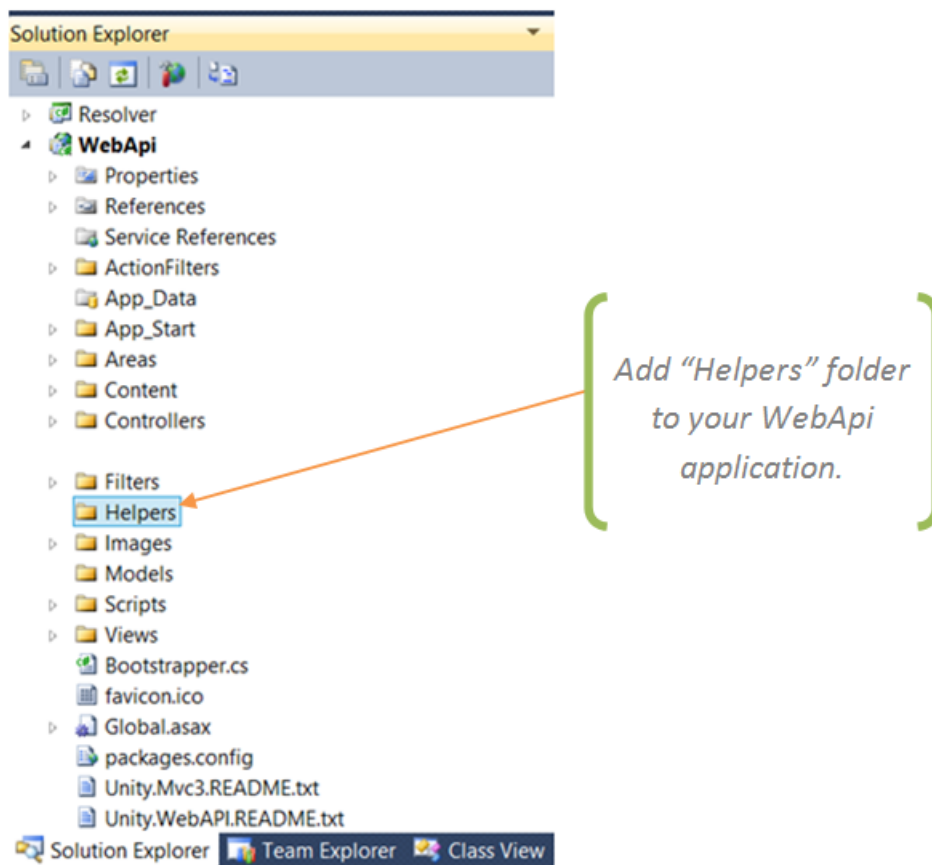
As mentioned in above target path, I have also created to “APILog” folder in the base directory of application –



Now we have configured the NLog in our application, and it is ready to start work for request logging. Note that in the rules section we have defined rules for logging in files as well as in windows events log as well, you can choose both of them or can opt for one too. Let's start with logging request in application, with action filters –

NLogger Class

Add a folder “Helpers” in the API, which will segregate the application code for readability, better understanding and maintainability.



To start add our main class “NLogger”, which will responsible for all types of errors and info logging, to same Helper folder. Here NLogger class implements ITraceWriter interface, which provides “Trace” method for the service request –

#region Using namespaces.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http.Tracing;

using NLog;

using System.Net.Http;


```
using System.Text;
```

```
using WebApi.ErrorHelper;
```

```
#endregion
```

```
namespace WebApi.Helpers
```

```
{
```

```
    /// <summary>
```

```
    /// Public class to log Error/info messages to the access log file
```

```
    /// </summary>
```

```
    public sealed class NLogger : ITraceWriter
```

```
    {
```

```
        #region Private member variables.
```

```
        private static readonly Logger ClassLogger = LogManager.GetCurrentClassLogger();
```

```
        private static readonly Lazy<Dictionary<TraceLevel, Action<string>>> LoggingMap =
new Lazy<Dictionary<TraceLevel, Action<string>>>(() => new Dictionary<TraceLevel,
Action<string>> { { TraceLevel.Info, ClassLogger.Info }, { TraceLevel.Debug,
ClassLogger.Debug }, { TraceLevel.Error, ClassLogger.Error }, { TraceLevel.Fatal,
ClassLogger.Fatal }, { TraceLevel.Warn, ClassLogger.Warn } });
```

```
        #endregion
```

```
        #region Private properties.
```

```
        /// <summary>
```

```
        /// Get property for Logger
```



```

    /// </summary>

    private Dictionary<TraceLevel, Action<string>> Logger
    {
        get { return LoggingMap.Value; }
    }

#endregion

#region Public member methods.

    /// <summary>
    /// Implementation of TraceWriter to trace the logs.
    /// </summary>
    /// <param name="request"></param>
    /// <param name="category"></param>
    /// <param name="level"></param>
    /// <param name="traceAction"></param>

    public void Trace(HttpRequestMessage request, string category, TraceLevel level,
    Action<TraceRecord> traceAction)
    {
        if (level != TraceLevel.Off)
        {
            if (traceAction != null && traceAction.Target != null)
            {

```



```

        category = category + Environment.NewLine + "Action Parameters : " +
traceAction.Target.ToJSON();

```

```

    }

```

```

    var record = new TraceRecord(request, category, level);

```

```

    if (traceAction != null) traceAction(record);

```

```

    Log(record);

```

```

}

```

```

}

```

```

#endregion

```

```

#region Private member methods.

```

```

/// <summary>

```

```

/// Logs info/Error to Log file

```

```

/// </summary>

```

```

/// <param name="record"></param>

```

```

private void Log(TraceRecord record)

```

```

{

```

```

    var message = new StringBuilder();

```

```

    if (!string.IsNullOrEmpty(record.Message))

```

```

        message.Append("").Append(record.Message + Environment.NewLine);

```

```

    if (record.Request != null)

```



```

{

    if (record.Request.Method != null)

        message.Append("Method: " + record.Request.Method +
Environment.NewLine);

    if (record.Request.RequestUri != null)

        message.Append("").Append("URL: " + record.Request.RequestUri +
Environment.NewLine);

    if (record.Request.Headers != null && record.Request.Headers.Contains("Token")
&& record.Request.Headers.GetValues("Token") != null &&
record.Request.Headers.GetValues("Token").FirstOrDefault() != null)

        message.Append("").Append("Token: " +
record.Request.Headers.GetValues("Token").FirstOrDefault() + Environment.NewLine);

}

if (!string.IsNullOrEmpty(record.Category))

    message.Append("").Append(record.Category);

if (!string.IsNullOrEmpty(record.Operator))

    message.Append(" ").Append(record.Operator).Append("
").Append(record.Operation);

Logger[record.Level](Convert.ToString(message) + Environment.NewLine);

```



```
    }  
  
    #endregion  
  
}  
  
}
```

Adding Action Filter

Action filter will be responsible for handling all the incoming requests to our APIs and logging them using NLogger class. We have “OnActionExecuting” method that is implicitly called if we mark our controllers or global application to use that particular filter. So each time any action of any controller will be hit, our “OnActionExecuting” method will execute to log the request.

Step 1: Adding `LoggingFilterAttribute` class

Create a class `LoggingFilterAttribute` to “ActionFilters” folder and add following code -

```
using System;  
  
using System.Collections.Generic;  
  
using System.Linq;  
  
using System.Web;  
  
using System.Web.Http.Filters;  
  
using System.Web.Http.Controllers;  
  
using System.Web.Http.Tracing;  
  
using System.Web.Http;  
  
using WebApi.Helpers;
```



```

namespace WebApi.ActionFilters
{
    public class LoggingFilterAttribute : ActionFilterAttribute
    {
        public override void OnActionExecuting(HttpContext filterContext)
        {
            GlobalConfiguration.Configuration.Services.Replace(typeof(ITraceWriter), new
NLogger());

            var trace = GlobalConfiguration.Configuration.Services.GetTraceWriter();

            trace.Info(filterContext.Request, "Controller : " +
filterContext.ControllerContext.ControllerDescriptor.ControllerType.FullName +
Environment.NewLine + "Action : " + filterContext.ActionDescriptor.ActionName, "JSON",
filterContext.ActionArguments);
        }
    }
}

```

The `LoggingFilterAttribute` class derived from `ActionFilterAttribute`, which is under `System.Web.Http.Filters` and overriding the `OnActionExecuting` method.

Here I have replaced the default “ITraceWriter” service with our `NLogger` class instance in the controller’s service container. Now `GetTraceWriter()` method will return our instance (instance `NLogger` class) and `Info()` will call `trace()` method of our `NLogger` class.

Note that the code below,


```
GlobalConfiguration.Configuration.Services.Replace(typeof(ITraceWriter), new NLogger());
```

is used to resolve dependency between ITaceWriter and NLogger class. Thereafter we use a variable named trace to get the instance and trace.Info() is used to log the request and whatever text we want to add along with that request.

Step 2: Registering Action Filter (LoggingFilterAttribute)

In order to register the created action filter to application's filters, just add a new instance of your action filter to config.Filters in `WebApiConfig` class.

```
using System.Web.Http;
```

```
using WebApi.ActionFilters;
```

```
namespace WebApi.App_Start
```

```
{
```

```
    public static class WebApiConfig
```

```
    {
```

```
        public static void Register(HttpConfiguration config)
```

```
        {
```

```
            config.Filters.Add(new LoggingFilterAttribute());
```

```
        }
```

```
    }
```



```
}
```

Now this action filter is applicable to all the controllers and actions in our project. You may not believe but request logging is done. It's time to run the application and validate our homework.



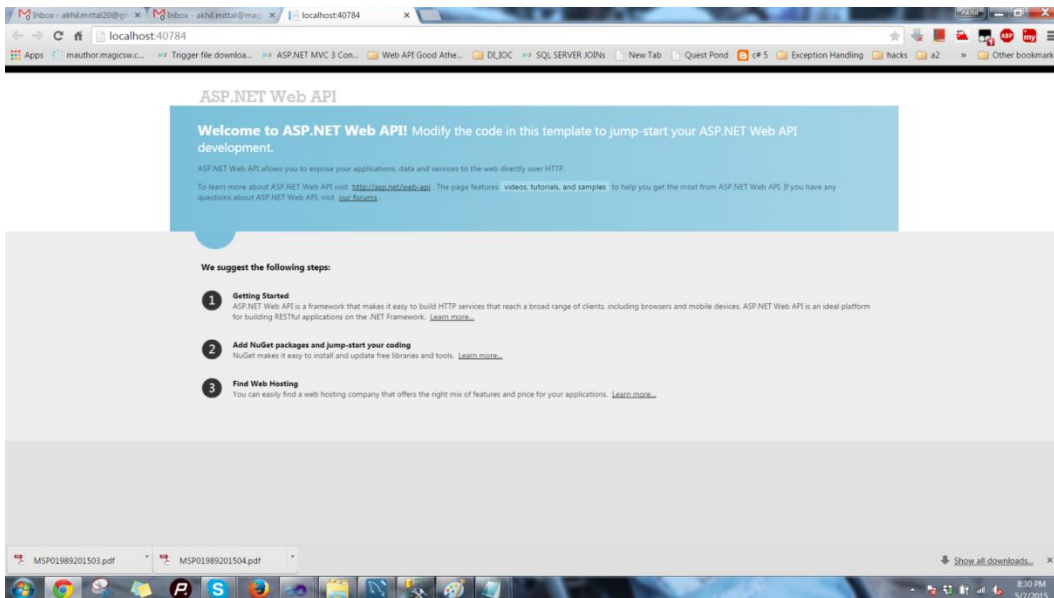
Image credit: <https://pixabay.com/en/social-media-network-media-54536/>

Running the application

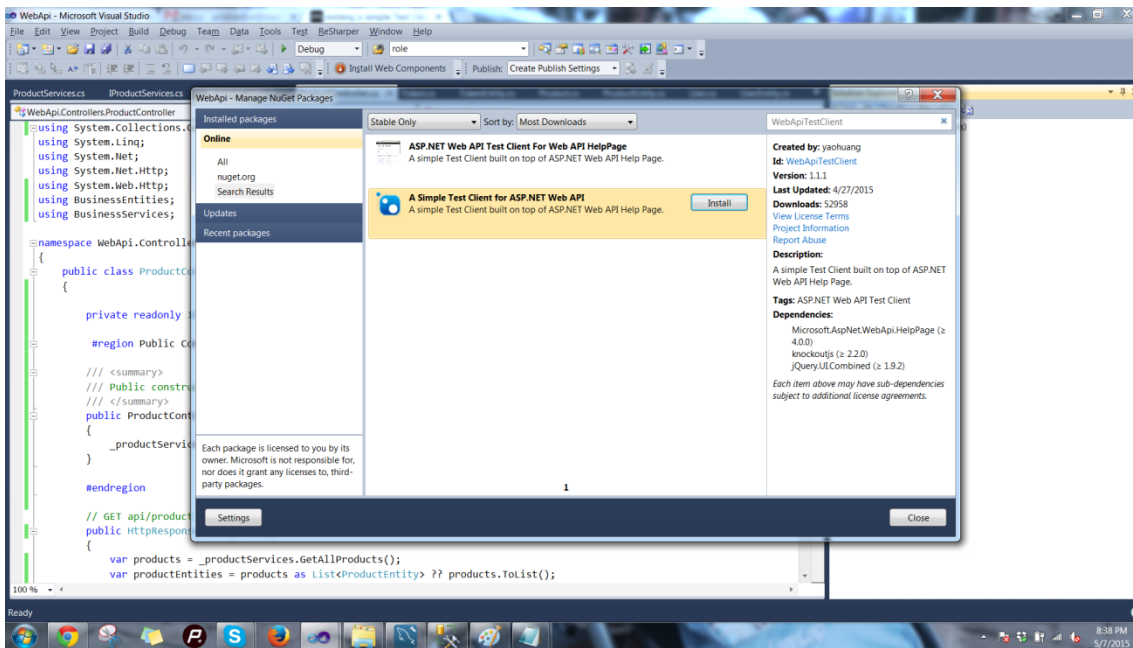
Let's run the application and try to make a call, using token based authorization, we have already covered authorization in day#5. You first need to authenticate your request using

login service , and then that service will return a token for making calls to other services. Use that token to make calls to other services, for more details you can read day5 of this series.

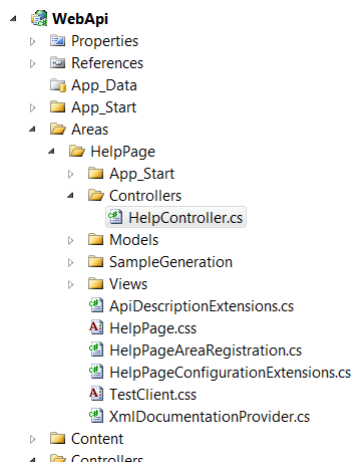
Just run the application, we get,



We already have our test client added, but for new readers, just go to Manage Nuget Packages, by right clicking WebAPI project and type WebAPITestClient in searchbox in online packages,



You'll get "A simple Test Client for ASP.NET Web API", just add it. You'll get a help controller in Areas-> HelpPage like shown below,



I have already provided the database scripts and data in my previous article, you can use the same.

Append "/help" in the application url, and you'll get the test client,

GET:

Product

API	Description
GET v1/Products/Product/all	Documentation for 'Get'.
GET v1/Products/Product/all?id={id}	Documentation for 'Get'.
GET v1/Products/Product/allproducts	Documentation for 'Get'.
GET v1/Products/Product/allproducts?id={id}	Documentation for 'Get'.
GET v1/Products/Product/myproduct/{id}	Documentation for 'Get'.
GET v1/Products/Product/particularproduct	Documentation for 'Get'.
GET v1/Products/Product/particularproduct/{id}	Documentation for 'Get'.
GET v1/Products/Product/productid	Documentation for 'Get'.
GET v1/Products/Product/productid/{id}	Documentation for 'Get'.

POST:

POST v1/Products/Product/Create	Documentation for 'Post'.
POST v1/Products/Product/Register	Documentation for 'Post'.

PUT:

[PUT v1/Products/Product/Update/productid/{id}](#)

Documentation for 'Put'.

[PUT v1/Products/Product/Modify/productid/{id}](#)

Documentation for 'Put'.

DELETE:

[DELETE v1/Products/Product/delete/productid/{id}](#)

Documentation for 'Delete'.

[DELETE v1/Products/Product/remove/productid/{id}](#)

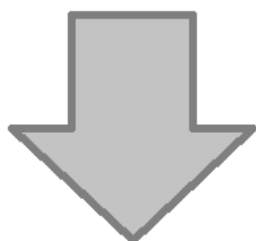
Documentation for 'Delete'.

[DELETE v1/Products/Product/clear/productid/{id}](#)

Documentation for 'Delete'.

You can test each service by clicking on it. Once you click on the service link, you'll be redirected to test the service page of that particular service. On that page there is a button Test API in the right bottom corner, just press that button to test your service,

**Press this button to
test the API**



A rectangular button with a light gray background and a thin black border. The text 'Test API' is centered on the button in a dark gray font.

Service for Get All products,

[Help](#) [Page](#) [Home](#)

GET v1/Products/Product/all

Documentation for 'Get'.

GET v1/Products/Product/all ✕

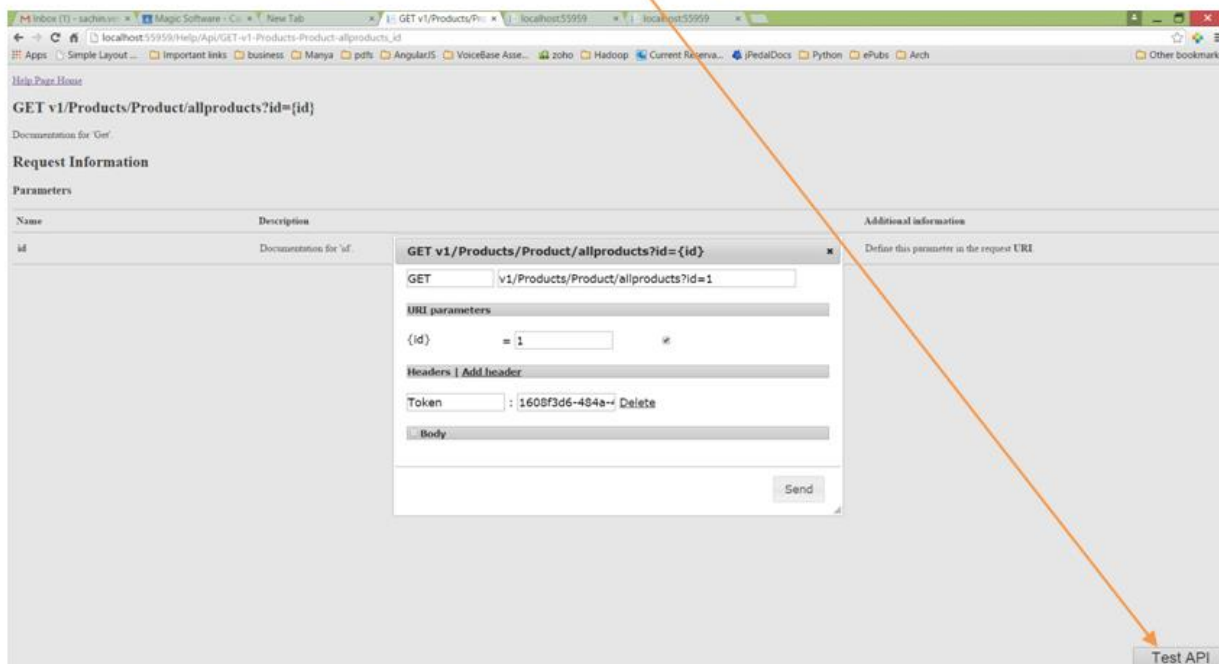
GET

Headers | [Add header](#)

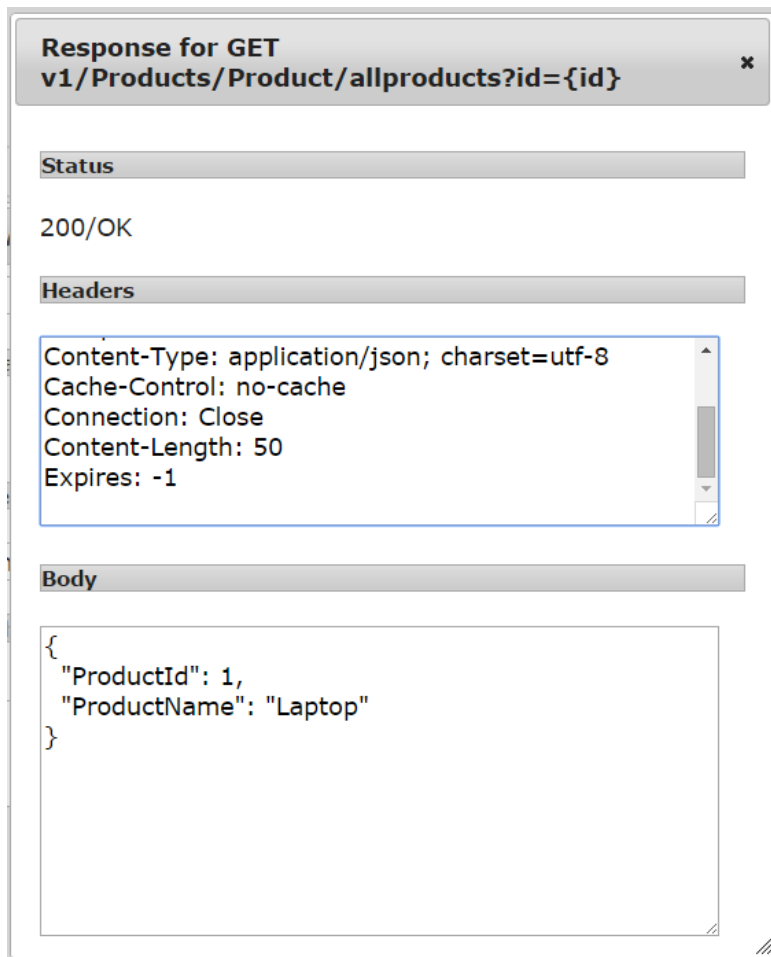
☐ **Body**

In below case, I have already generated the token and now using it to make call to fetch all the products from products table in database.

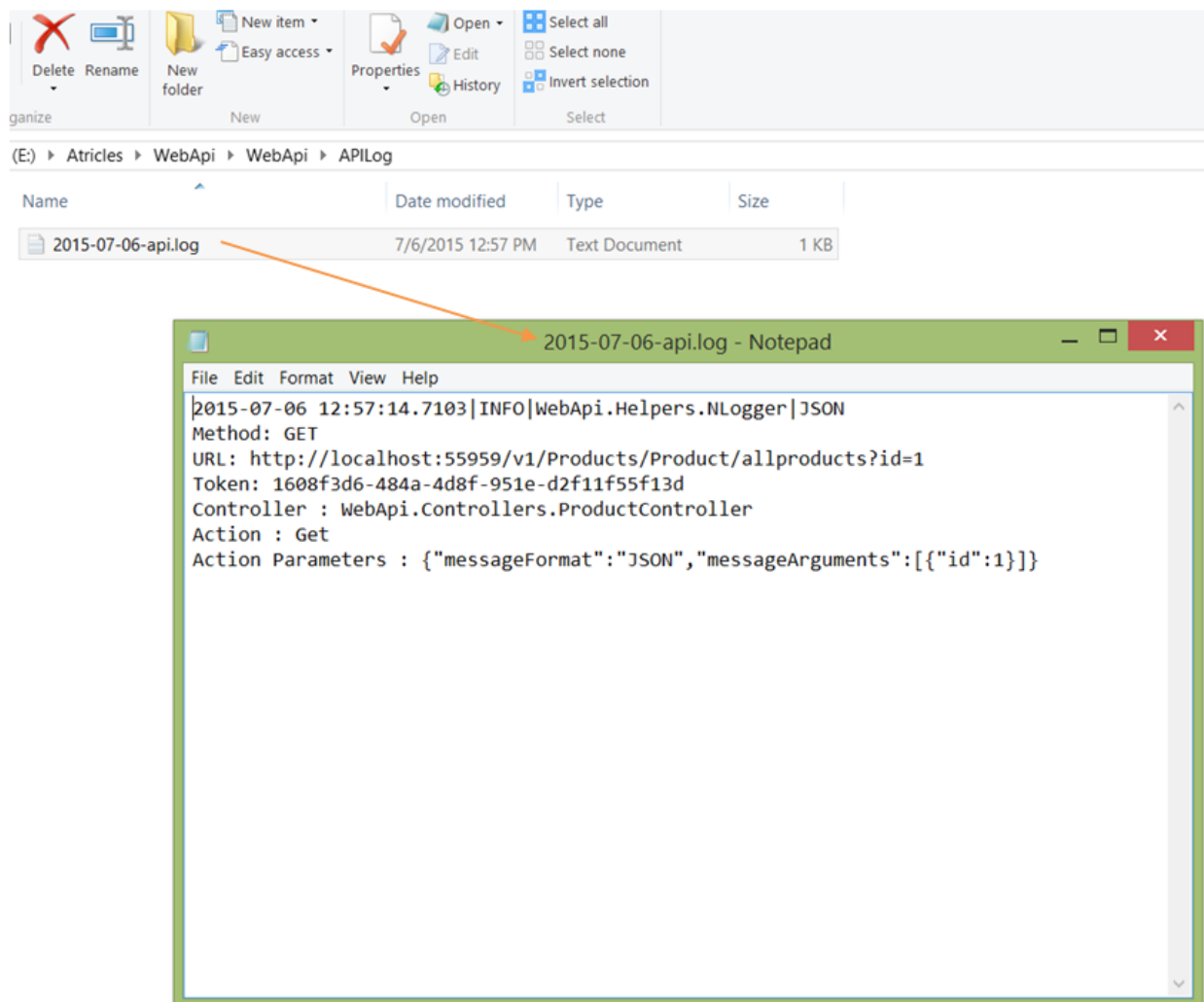
Launch the testing form by clicking Test API button.



Here I have called allproducts API, Add the value for parameter Id and “Token” header with its current value and click to get the result -



Now let's see what happens to our APILog folder in application. Here you find the API log has been created, with the same name we have configured in NLog configuration in web.config file. The log file contains all the supplied details like Timestamp, Method type, URL , Header information (Token), Controller name, action and action parameters. You can also add more details to this log which you deem important for your application.



Logging Done!



Exception Logging

Our logging setup is completed, now we'll focus on centralizing exception logging as well, so that none of the exception escapes without logging itself. Logging exception is of very high importance, it keeps track of all the exceptions. No matter business or application or system exceptions, but all of them have to be logged.

Implementing Exception logging

Step 1: Exception Filter Attribute

Adding an action filter in our application for logging the exceptions, for this create a class `GlobalExceptionHandler` to "ActionFilter" folder and add the code below, the class is derived from `ExceptionHandlerAttribute`, which is under `System.Web.Http.Filters`.

I have override OnException() method, and replaced the default “ITraceWriter” service with our NLogger class instance in the controller’s service container, same as we have done in Action logging in above section. Now GetTraceWriter() method will return our instance (instance NLogger class) and Info() will call trace() method of NLogger class.

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Http.Filters;

using System.Web.Http;

using System.Web.Http.Tracing;

using WebApi.Helpers;

using System.ComponentModel.DataAnnotations;

using System.Net.Http;

using System.Net;

namespace WebApi.ActionFilters
{
    /// <summary>
    /// Action filter to handle for Global application errors.
    /// </summary>
```



```

public class GlobalExceptionHandler : ExceptionFilterAttribute
{
    public override void OnException(HttpActionExecutedContext context)
    {
        GlobalConfiguration.Configuration.Services.Replace(typeof(ITraceWriter), new
NLogger());

        var trace = GlobalConfiguration.Configuration.Services.GetTraceWriter();

        trace.Error(context.Request, "Controller : " +
context.ActionContext.ControllerContext.ControllerDescriptor.ControllerType.FullName +
Environment.NewLine + "Action : " + context.ActionContext.ActionDescriptor.ActionName,
context.Exception);

        var exceptionType = context.Exception.GetType();

        if (exceptionType == typeof(ValidationException))
        {
            var resp = new HttpResponseMessage(HttpStatusCode.BadRequest) { Content =
new StringContent(context.Exception.Message), ReasonPhrase = "ValidationException", };

            throw new HttpResponseException(resp);

        }

        else if (exceptionType == typeof(UnauthorizedAccessException))
        {
            throw new
HttpResponseException(context.Request.CreateResponse(HttpStatusCode.Unauthorized));

```



```

    }

    else

    {

        throw new
        HttpResponseException(context.Request.CreateResponse(HttpStatusCode.InternalServerError));

    }

}

}

}

```

Step 2: Modify NLogger Class

Our NLogger class is capable to log all info and events, I have done some changes in private method Log() to handle the exceptions –

[#region](#) Private member methods.

```

/// <summary>

/// Logs info/Error to Log file

/// </summary>

/// <param name="record"></param>

private void Log(TraceRecord record)

{

    var message = new StringBuilder();

```



```

if (!string.IsNullOrEmpty(record.Message))

    message.Append("").Append(record.Message + Environment.NewLine);

if (record.Request != null)
{

    if (record.Request.Method != null)

        message.Append("Method: " + record.Request.Method + Environment.NewLine);

    if (record.Request.RequestUri != null)

        message.Append("").Append("URL: " + record.Request.RequestUri +
Environment.NewLine);

    if (record.Request.Headers != null && record.Request.Headers.Contains("Token")
&& record.Request.Headers.GetValues("Token") != null &&
record.Request.Headers.GetValues("Token").FirstOrDefault() != null)

        message.Append("").Append("Token: " +
record.Request.Headers.GetValues("Token").FirstOrDefault() + Environment.NewLine);

}

if (!string.IsNullOrEmpty(record.Category))

    message.Append("").Append(record.Category);

if (!string.IsNullOrEmpty(record.Operator))

```



```

        message.Append(" ").Append(record.Operator).Append("
").Append(record.Operation);

        if (record.Exception != null &&
!string.IsNullOrEmpty(record.Exception.GetBaseException().Message))
        {
            var exceptionType = record.Exception.GetType();

            message.Append(Environment.NewLine);

            message.Append("").Append("Error: " +
record.Exception.GetBaseException().Message + Environment.NewLine);
        }

        Logger[record.Level](Convert.ToString(message) + Environment.NewLine);
    }

```

Step 3: Modify Controller for Exceptions

Our application is now ready to run, but there is no exception in our code, so I added a throw exception code in ProductController, just the Get(int id) method so that it can throw exception for testing our exception logging mechanism, It will throw an exception if the product is not there in database with the provided id.

```
// GET api/product/5
```



```
[GET("productid/{id?}")]
```

```
[GET("particularproduct/{id?}")]
```

```
[GET("myproduct/{id:range(1, 3)}")]
```

```
public HttpResponseMessage Get(int id)
```

```
{
```

```
var product = _productServices.GetProductById(id);
```

```
    if (product != null)
```

```
        return Request.CreateResponse(HttpStatusCode.OK, product);
```

```
throw new Exception("No product found for this id");
```

```
    //return Request.CreateErrorResponse(HttpStatusCode.NotFound, "No product found  
for this id");
```

```
}
```

Step 4: Run the application

Run the application and click on Product/all API

ASP.NET Web API Help Page

Introduction

Provide a general description of your APIs here.

Authenticate

API
POST get/token
POST authenticate
POST login

Product

API
GET v1/Products/Product/allproducts
GET v1/Products/Product/allproducts?id={id}
GET v1/Products/Product/all
GET v1/Products/Product/all?id={id}
GET v1/Products/Product/myproduct/{id}

GET v1/Products/Product/all?id={id}

GET

v1/Products/Product/all?id=1

URI parameters

{id}

=

1

☒

Headers | Add header

Token

:

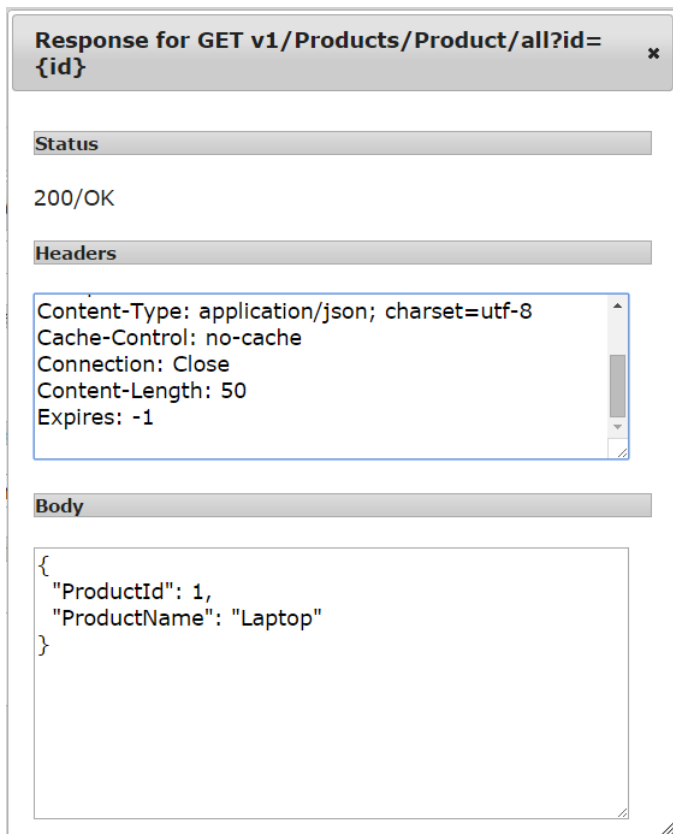
1608f3d6-484a-4

Delete

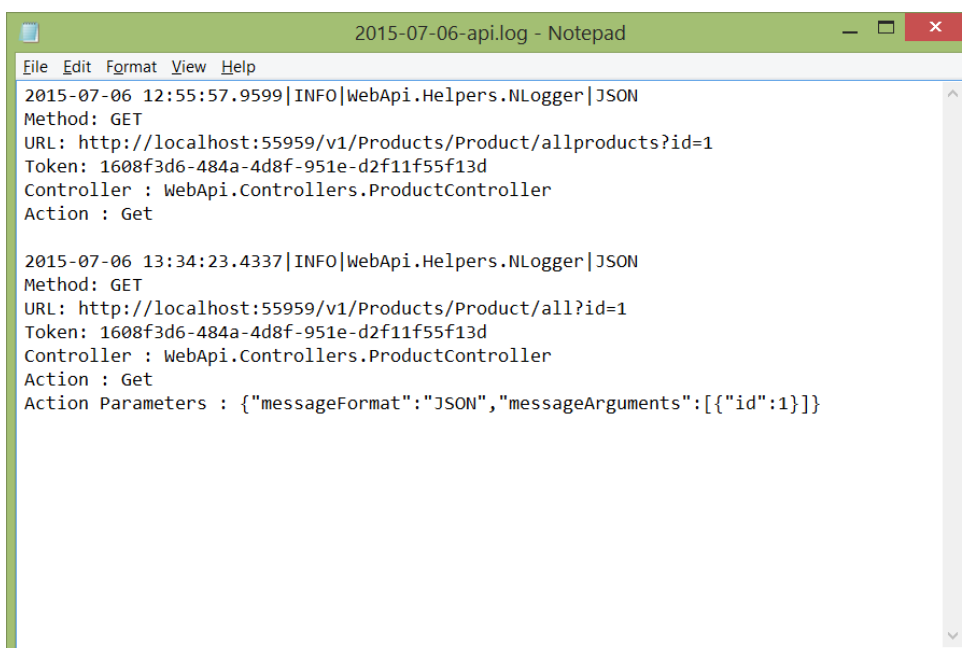
☐ Body

Send

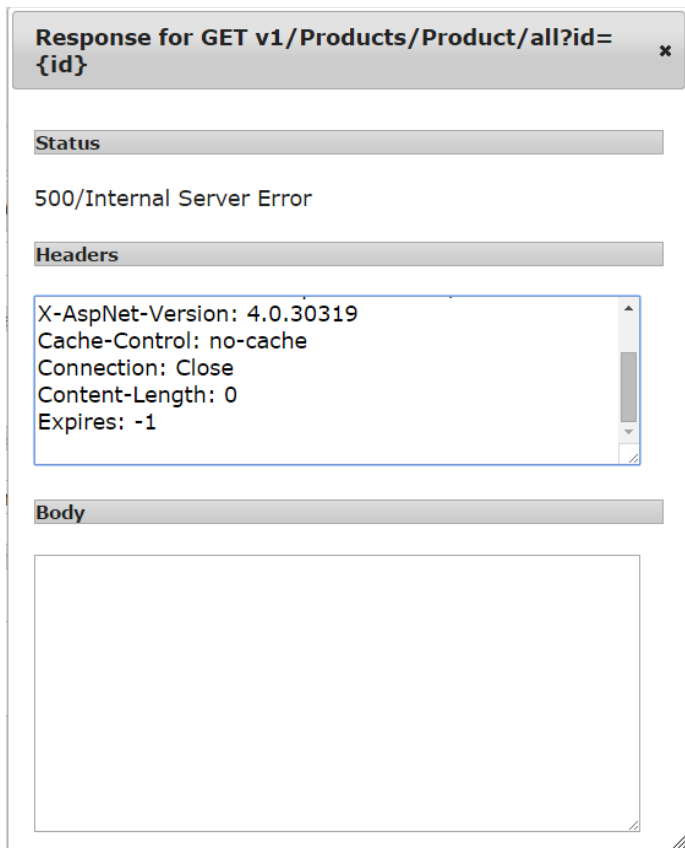
Add the parameter id value to 1 and header Token with it's current value, click on send button to get the result –



Now we can see that the Status is 200/OK, and we also get a product with the provided id in the response body. Let's see the API log now –



The log has captured the call of Product API, now provide a new product id as parameter, which is not there in database, I am using 12345 as product id and result is –



We can see there is an 500/Internal Server Error now in response status, let's check the API Log-

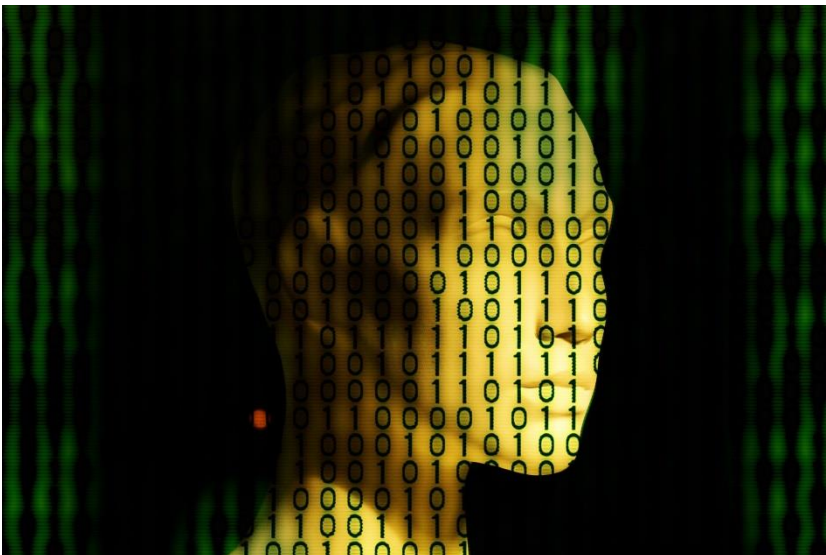

```
2015-07-06-api.log - Notepad
File Edit Format View Help
2015-07-06 12:55:57.9599|INFO|WebApi.Helpers.NLogger|JSON
Method: GET
URL: http://localhost:55959/v1/Products/Product/allproducts?id=1
Token: 1608f3d6-484a-4d8f-951e-d2f11f55f13d
Controller : WebApi.Controllers.ProductController
Action : Get

2015-07-06 13:34:23.4337|INFO|WebApi.Helpers.NLogger|JSON
Method: GET
URL: http://localhost:55959/v1/Products/Product/all?id=1
Token: 1608f3d6-484a-4d8f-951e-d2f11f55f13d
Controller : WebApi.Controllers.ProductController
Action : Get
Action Parameters : {"messageFormat":"JSON","messageArguments":[{"id":1}]}

2015-07-06 13:39:59.7784|INFO|WebApi.Helpers.NLogger|JSON
Method: GET
URL: http://localhost:55959/v1/Products/Product/all?id=12345
Token: 1608f3d6-484a-4d8f-951e-d2f11f55f13d
Controller : WebApi.Controllers.ProductController
Action : Get
Action Parameters : {"messageFormat":"JSON","messageArguments":[{"id":12345}]}

2015-07-06 13:40:05.2033|ERROR|WebApi.Helpers.NLogger|Method: GET
URL: http://localhost:55959/v1/Products/Product/all?id=12345
Token: 1608f3d6-484a-4d8f-951e-d2f11f55f13d
Controller : WebApi.Controllers.ProductController
Action : Get
Action Parameters :
Error: No product found for this id
```

Well, now the log has captured both the event and error of same call on the server, you can see call log details and the error with provided error message in the log.

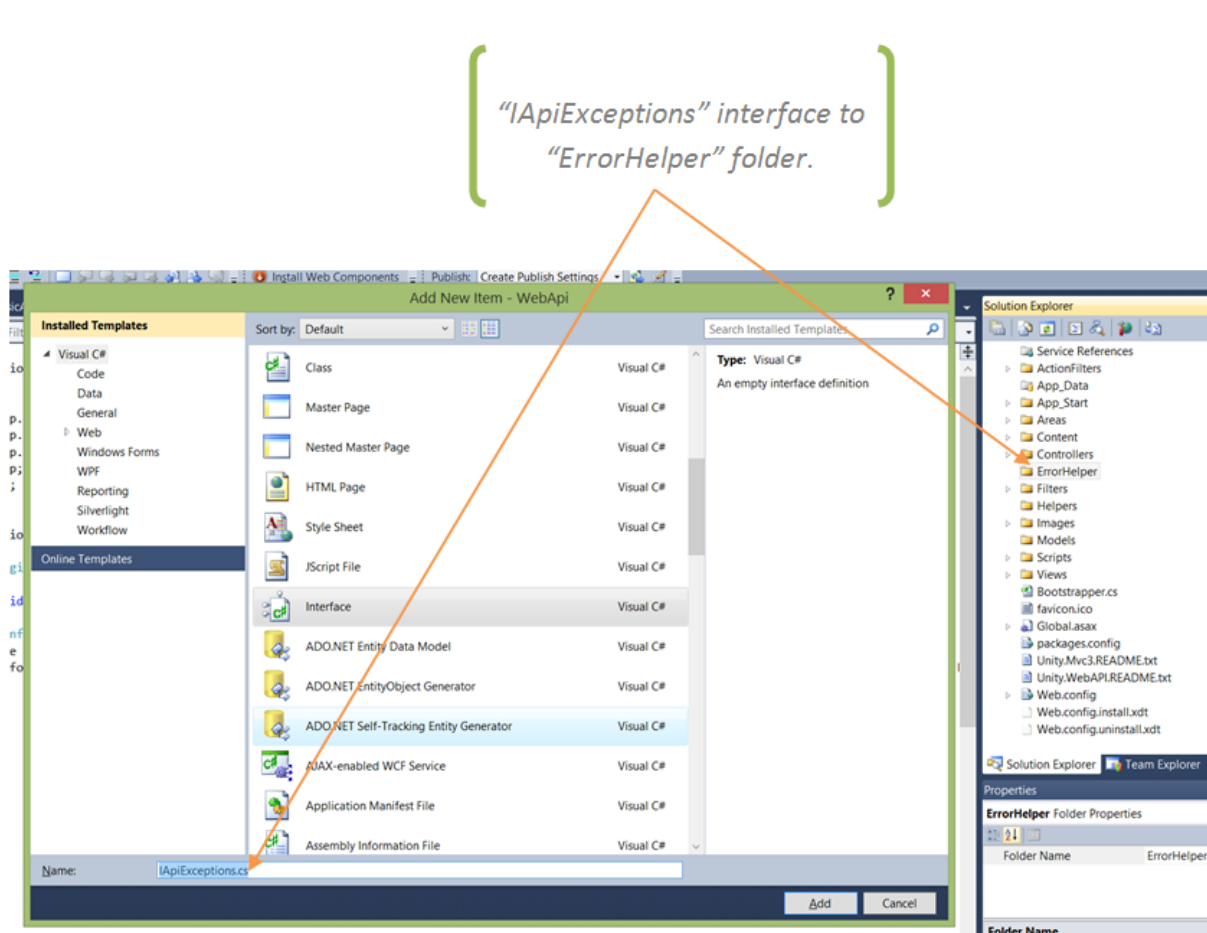


Custom Exception logging

In the above section we have implemented exception logging, but there is default system response and status (i. e. 500/Internal Server Error) , It will be always good to have your own custom response and exceptions for your API. That will be easier for client to consume and understand the API responses.

Step 1: Add Custom Exception Classes

Add “Error Helper” folder to application to maintain our custom exception classes separately and add “IApiExceptions” interface to newly created “ErrorHandler” folder -



Add following code the [IApiExceptions](#) interface, this will serve as a template for all exception classes, I have added four common properties for our custom classes to maintain Error Code, ErrorDescription, HttpStatus (Contains the values of status codes defined for HTTP) and ReasonPhrase.


```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Net;


namespace WebApi.ErrorHelper
{
    /// <summary>
    /// IApiExceptions Interface
    /// </summary>
    public interface IApiExceptions
    {
        /// <summary>
        /// ErrorCode
        /// </summary>
        int ErrorCode { get; set; }

        /// <summary>
        /// ErrorDescription
        /// </summary>
        string ErrorDescription { get; set; }

        /// <summary>
```



```

    /// HttpStatus

    /// </summary>

    HttpStatusCode HttpStatus { get; set; }

    /// <summary>

    /// ReasonPhrase

    /// </summary>

    string ReasonPhrase { get; set; }

}

}

```

Here, I divided our exceptions in three categories

1. API Exceptions – for API level exceptions.
2. Business Exceptions – for exceptions at business logic level.
3. Data Exceptions – Data related exceptions.

To implement this create a three new classes ApiException.cs, ApiDataException.cs and ApiBusinessException classes to same folder which implements IApiExceptions interface with following code to the classes –

#region Using namespaces.

using System;

using System.Net;

using System.Runtime.Serialization;


```
#endregion
```

```
namespace WebApi.ErrorHelper
{
    /// <summary>
    /// Api Exception
    /// </summary>
    [Serializable]
    [DataContract]
    public class ApiException : Exception, IApiExceptions
    {
        #region Public Serializable properties.
        [DataMember]
        public int ErrorCode { get; set; }

        [DataMember]
        public string ErrorDescription { get; set; }

        [DataMember]
        public HttpStatusCode HttpStatus { get; set; }

        string reasonPhrase = "ApiException";

        [DataMember]
```



```

    public string ReasonPhrase
    {
        get { return this.reasonPhrase; }

        set { this.reasonPhrase = value; }
    }
}
#endregion
}
}

```

I have initialized ReasonPhrase property with different default values in these classes to differentiate the implementation, you can use implement your custom classes as per your application needs.

The directives applied on class as `Serializable` and `DataContract` to make sure that the class defines or implements a data contract is serializable and can be serialize by a serializer.

Note – Add reference of “System.Runtime.Serialization.dll” dll if you facing any assembly issue.

In the same way add “ApiBusinessException” and “ApiDataException” classes into the same folder, with the following code –

`#region` Using namespaces.


```
using System;

using System.Net;

using System.Runtime.Serialization;

#endregion


namespace WebApi.ErrorHelper
{
    /// <summary>
    /// Api Business Exception
    /// </summary>
    [Serializable]
    [DataContract]

    public class ApiBusinessException : Exception, IApiExceptions
    {
        #region Public Serializable properties.

        [DataMember]
        public int ErrorCode { get; set; }

        [DataMember]
        public string ErrorDescription { get; set; }

        [DataMember]
        public HttpStatusCode HttpStatus { get; set; }

        string reasonPhrase = "ApiBusinessException";
    }
}
```



```
[DataMember]
```

```
public string ReasonPhrase
```

```
{
```

```
    get { return this.reasonPhrase; }
```

```
    set { this.reasonPhrase = value; }
```

```
}
```

```
#endregion
```

```
#region Public Constructor.
```

```
/// <summary>
```

```
/// Public constructor for Api Business Exception
```

```
/// </summary>
```

```
/// <param name="errorCode"></param>
```

```
/// <param name="errorDescription"></param>
```

```
/// <param name="httpStatus"></param>
```

```
public ApiBusinessException(int errorCode, string errorDescription, HttpStatusCode  
httpStatus)
```

```
{
```

```
    ErrorCode = errorCode;
```

```
    ErrorDescription = errorDescription;
```

```
    HttpStatus = httpStatus;
```



```
    }  
  
    #endregion  
  
}  
  
}
```

#region Using namespaces.

```
using System;  
  
using System.Net;  
  
using System.Runtime.Serialization;  
  
#endregion
```

namespace WebApi.ErrorHelper

```
{  
  
    /// <summary>  
    /// Api Data Exception  
    /// </summary>  
  
    [Serializable]  
  
    [DataContract]  
  
    public class ApiDataException : Exception, IApiExceptions  
  
    {  
  
        #region Public Serializable properties.  
  
        [DataMember]
```



```
public int ErrorCode { get; set; }

[DataMember]

public string ErrorDescription { get; set; }

[DataMember]

public HttpStatusCode HttpStatus { get; set; }

string reasonPhrase = "ApiDataException";

[DataMember]

public string ReasonPhrase
{
    get { return this.reasonPhrase; }

    set { this.reasonPhrase = value; }
}

#endregion

#region Public Constructor.

/// <summary>
/// Public constructor for Api Data Exception
/// </summary>
/// <param name="errorCode"></param>
```



```

    /// <param name="errorDescription"></param>

    /// <param name="httpStatus"></param>

    public ApiDataException(int errorCode, string errorDescription, HttpStatusCode
httpStatus)

    {

        ErrorCode = errorCode;

        ErrorDescription = errorDescription;

        HttpStatus = httpStatus;

    }

    #endregion

}

}

```

Json Serializers

There are some objects need to be serialized in json, to log and to transfer through the modules, for this I have add some extension methods to Object class –

For that add “*System.Web.Extensions.dll*” reference to project and add “JSONHelper” class to Helpers folder, with following code –

```

#region Using namespaces.

using System.Web.Script.Serialization;

using System.Data;

```



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

```
using System;
```

```
#endregion
```

```
namespace WebApi.Helpers
```

```
{  
  
    public static class JSONHelper  
    {  
  
        #region Public extension methods.  
  
        /// <summary>  
        /// Extened method of object class, Converts an object to a json string.  
        /// </summary>  
        /// <param name="obj"></param>  
        /// <returns></returns>  
  
        public static string ToJSON(this object obj)  
        {  
  
            var serializer = new JavaScriptSerializer();  
  
            try  
            {  
  
                return serializer.Serialize(obj);  
  
            }  
  
            catch(Exception ex)
```



```

        {
            return "";
        }
    }

    #endregion
}
}

```

In the above code “ToJson()” method is an extension of base Object class, which serializes supplied the object to a Json string. The method using “JavaScriptSerializer” class which exist in “System.Web.Script.Serialization”.

Modify NLogger Class

For exception handling I have modified the Log() method of NLogger, which will now handle the different API exceptions.

```

/// <summary>
/// Logs info/Error to Log file
/// </summary>
/// <param name="record"></param>
private void Log(TraceRecord record)
{
    var message = new StringBuilder();

```



```

if (!string.IsNullOrEmpty(record.Message))

    message.Append("").Append(record.Message + Environment.NewLine);

if (record.Request != null)
{
    if (record.Request.Method != null)

        message.Append("Method: " + record.Request.Method +
Environment.NewLine);

    if (record.Request.RequestUri != null)

        message.Append("").Append("URL: " + record.Request.RequestUri +
Environment.NewLine);

    if (record.Request.Headers != null && record.Request.Headers.Contains("Token")
&& record.Request.Headers.GetValues("Token") != null &&
record.Request.Headers.GetValues("Token").FirstOrDefault() != null)

        message.Append("").Append("Token: " +
record.Request.Headers.GetValues("Token").FirstOrDefault() + Environment.NewLine);
}

if (!string.IsNullOrEmpty(record.Category))

    message.Append("").Append(record.Category);

if (!string.IsNullOrEmpty(record.Operator))

```



```

        message.Append(" ").Append(record.Operator).Append("
").Append(record.Operation);

        if (record.Exception != null &&
!string.IsNullOrEmpty(record.Exception.GetBaseException().Message))
        {
            var exceptionType = record.Exception.GetType();

            message.Append(Environment.NewLine);

            if (exceptionType == typeof(ApiException))
            {
                var exception = record.Exception as ApiException;

                if (exception != null)
                {
                    message.Append("").Append("Error: " + exception.ErrorDescription +
Environment.NewLine);

                    message.Append("").Append("Error Code: " + exception.ErrorCode +
Environment.NewLine);
                }
            }

            else if (exceptionType == typeof(ApiBusinessException))
            {
                var exception = record.Exception as ApiBusinessException;

                if (exception != null)
                {

```



```

        message.Append("").Append("Error: " + exception.ErrorDescription +
Environment.NewLine);

        message.Append("").Append("Error Code: " + exception.ErrorCode +
Environment.NewLine);

    }

}

else if (exceptionType == typeof(ApiDataException))
{
    var exception = record.Exception as ApiDataException;

    if (exception != null)
    {
        message.Append("").Append("Error: " + exception.ErrorDescription +
Environment.NewLine);

        message.Append("").Append("Error Code: " + exception.ErrorCode +
Environment.NewLine);

    }

}

else

    message.Append("").Append("Error: " +
record.Exception.GetBaseException().Message + Environment.NewLine);

}

Logger[record.Level](Convert.ToString(message) + Environment.NewLine);

}

```


The code above checks the exception object of `TraceRecord` and updates the logger as per the exception type.

Modify `GlobalExceptionHandler`

As we have created `GlobalExceptionHandler` to handle all exceptions and create response in case of any exception. Now I have added some new code to this in order to enable the `GlobalExceptionHandler` class to handle custom exceptions. I am adding only modified method here for your reference .

```
public override void OnException(HttpContext context)
{
    GlobalConfiguration.Configuration.Services.Replace(typeof(ILogger), new
    NLogger());

    var trace = GlobalConfiguration.Configuration.Services.GetTraceWriter();

    trace.Error(context.Request, "Controller : " +
context.ActionContext.ControllerContext.ControllerDescriptor.ControllerType.FullName +
Environment.NewLine + "Action : " + context.ActionContext.ActionDescriptor.ActionName,
context.Exception);

    var exceptionType = context.Exception.GetType();

    if (exceptionType == typeof(ValidationException))
    {
        var resp = new HttpResponseMessage(HttpStatusCode.BadRequest) { Content =
new StringContent(context.Exception.Message), ReasonPhrase = "ValidationException", };

        throw new HttpResponseException(resp);
    }
}
```



```

    }

    else if (exceptionType == typeof(UnauthorizedAccessException))
    {
        throw new
        HttpResponseMessage(context.Request.CreateResponse(HttpStatusCode.Unauthorized,
        new ServiceStatus() { StatusCode = (int)HttpStatusCode.Unauthorized, StatusMessage =
        "Unauthorized", ReasonPhrase = "Unauthorized Access" }));
    }

    else if (exceptionType == typeof(ApiException))
    {
        var webapiException = context.Exception as ApiException;

        if (webapiException != null)

            throw new
            HttpResponseMessage(context.Request.CreateResponse(webapiException.HttpStatus,
            new ServiceStatus() { StatusCode = webapiException.ErrorCode, StatusMessage =
            webapiException.ErrorDescription, ReasonPhrase = webapiException.ReasonPhrase }));
    }

    else if (exceptionType == typeof(ApiBusinessException))
    {
        var businessException = context.Exception as ApiBusinessException;

        if (businessException != null)

            throw new
            HttpResponseMessage(context.Request.CreateResponse(businessException.HttpStatus,
            new ServiceStatus() { StatusCode = businessException.ErrorCode, StatusMessage =
            businessException.ErrorDescription, ReasonPhrase = businessException.ReasonPhrase }));
    }

```



```

    }

    else if (exceptionType == typeof(ApiDataException))
    {
        var dataException = context.Exception as ApiDataException;

        if (dataException != null)

            throw new
HttpResponseException(context.Request.CreateResponse(dataException.HttpStatus, new
ServiceStatus() { StatusCode = dataException.ErrorCode, StatusMessage =
dataException.ErrorDescription, ReasonPhrase = dataException.ReasonPhrase }));

    }

    else

    {

        throw new
HttpResponseException(context.Request.CreateResponse(HttpStatusCode.InternalServerEr
ror));

    }

}

```

In the above code I have modified the overridden method OnException() and created new Http response exception based on the different exception types.

Modify Product Controller

Now modify the Product controller to throw our custom exception form, please look into the Get method I have modified to throw the APIDataException in case if data is not found and APIException in any other kind of error.


```
// GET api/product/5
```

```
[GET("productid/{id?}")]
```

```
[GET("particularproduct/{id?}")]
```

```
[GET("myproduct/{id:range(1, 3)}")]
```

```
public HttpResponseMessage Get(int id)
```

```
{
```

```
if (id != null)
```

```
{
```

```
    var product = _productServices.GetProductById(id);
```

```
    if (product != null)
```

```
        return Request.CreateResponse(HttpStatusCode.OK, product);
```

```
throw new ApiDataException(1001, "No product found for this id.",  
HttpStatusCode.NotFound);
```

```
}
```

```
throw new ApiException() { ErrorCode = (int)HttpStatusCode.BadRequest,  
ErrorDescription = "Bad Request..." };
```

```
}
```

Run the application

Run the application and click on Product/all API -

ASP.NET Web API Help Page

Introduction

Provide a general description of your APIs here.

Authenticate

API
POST get/token
POST authenticate
POST login

Product

API
GET v1/Products/Product/allproducts
GET v1/Products/Product/allproducts?id={id}
GET v1/Products/Product/all
GET v1/Products/Product/all?id={id}
GET v1/Products/Product/myproduct/{id}

GET v1/Products/Product/all?id={id} x

GET

v1/Products/Product/all?id=1

URI parameters

{id}

=

1

☒

Headers | Add header

Token

:

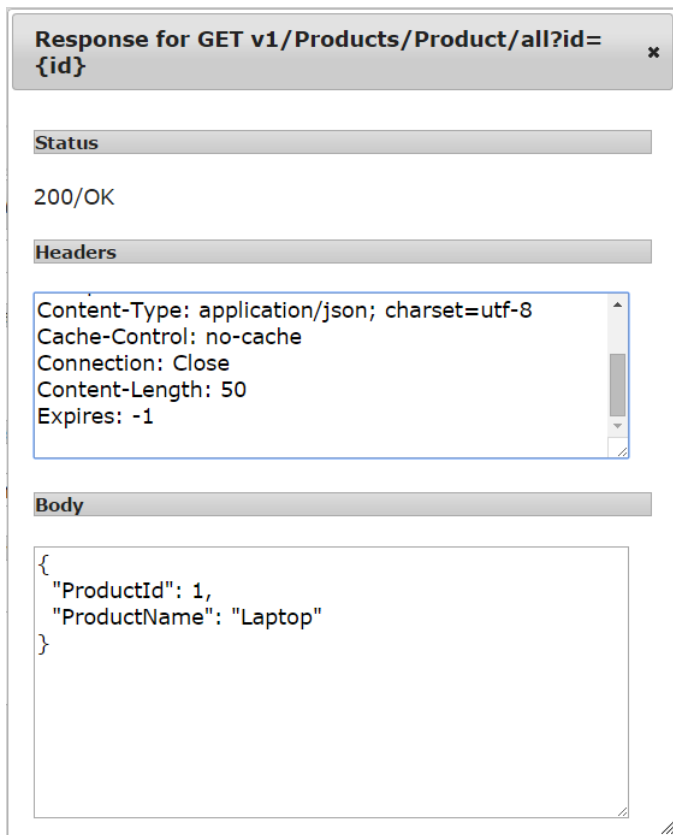
1608f3d6-484a-4

Delete

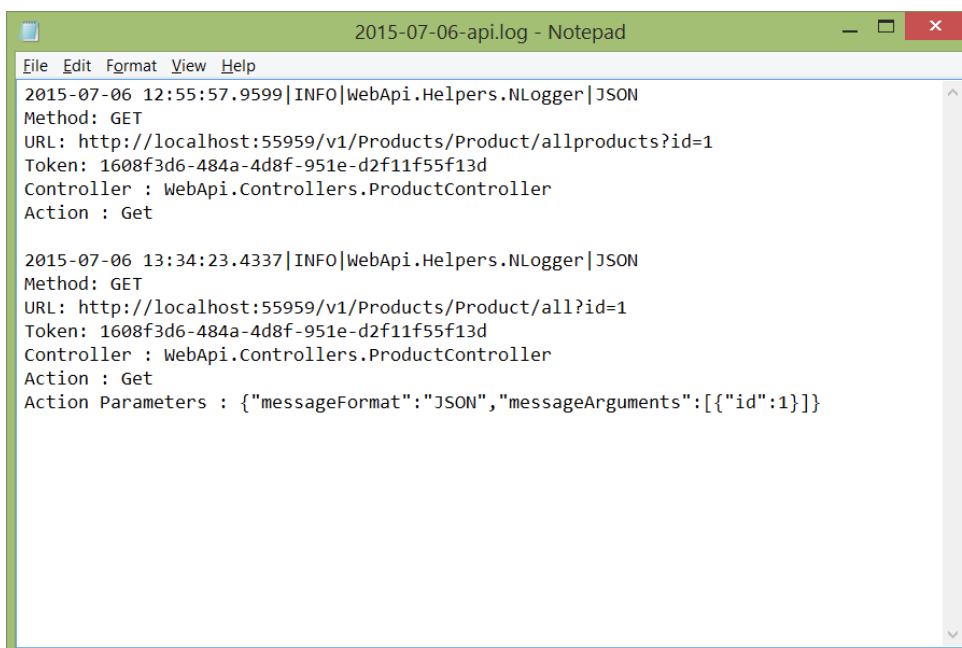
☐ Body

Send

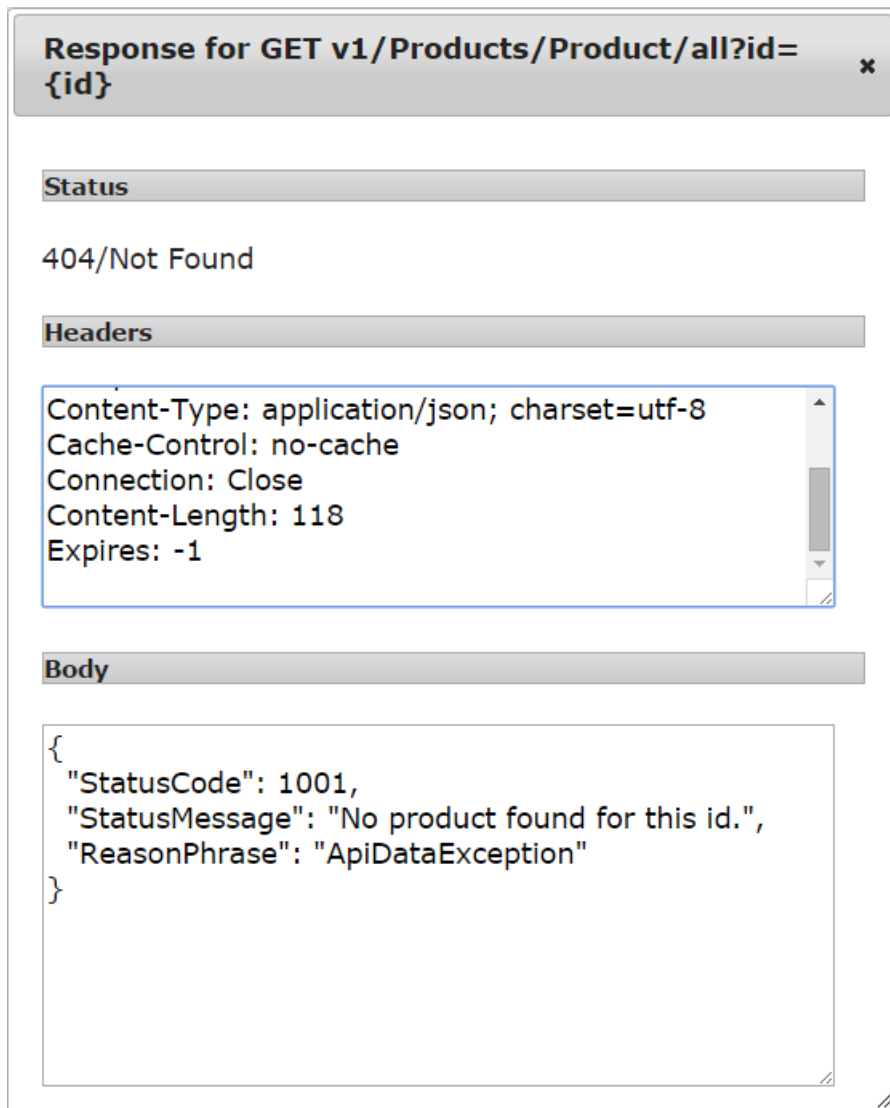
Add the parameter id value to 1 and header Token with its current value, click on send button to get the result –



Now we can see that the Status is 200/OK, and we also get a product with the provided id in the response body. Lets see the API log now –

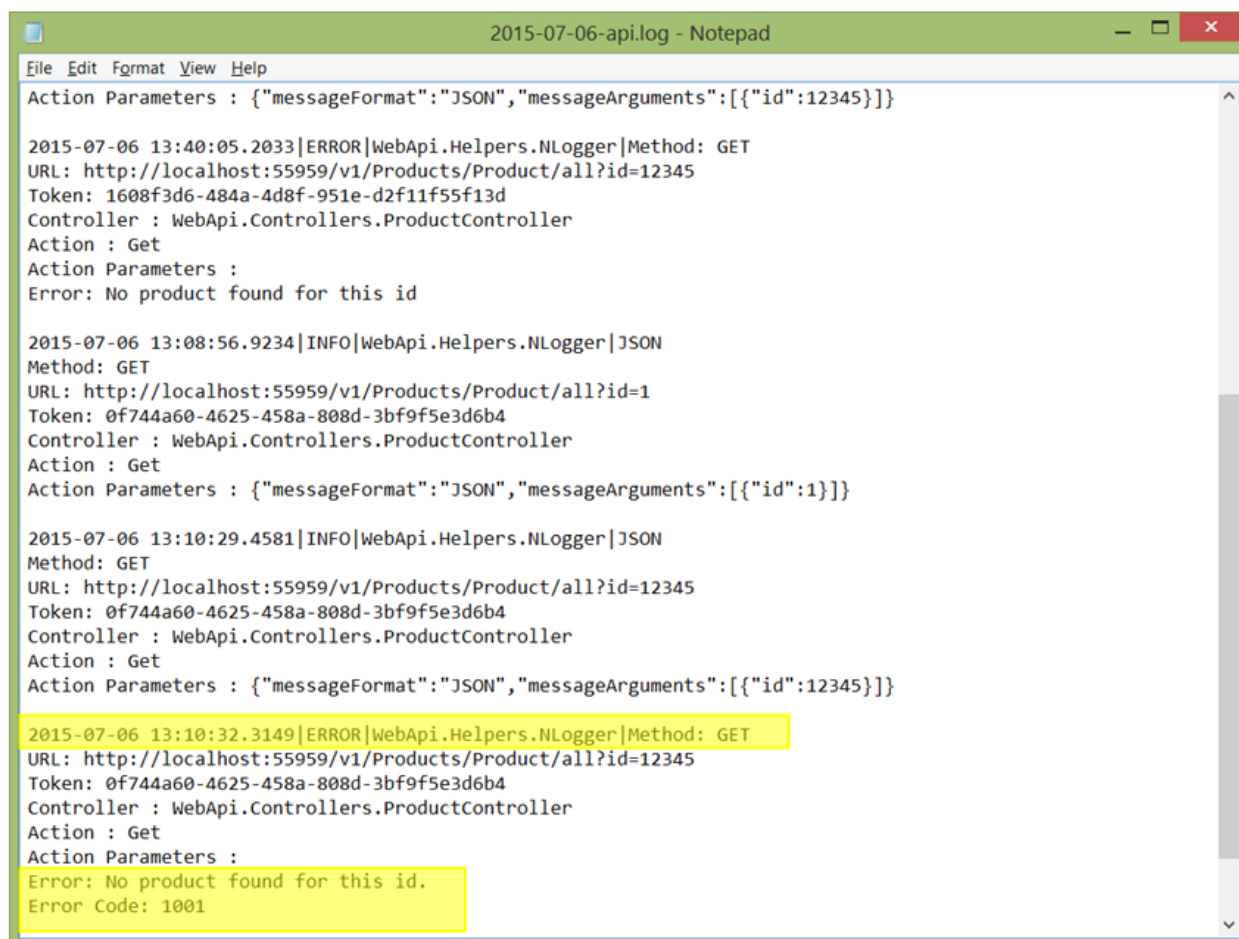


The log has captured the call of Product API, now provide a new product id as parameter, which is not there in database, I am using 12345 as product id and result is –



We can see, now there is a custom error status code “1001” and messages “No product found for this id.” And the generic status code “500/Internal Server Error” is now replaced with our supplied code “404/ Not Found”, which is more meaningful for the client or consumer.

Lets see the APILog now -



```
2015-07-06-api.log - Notepad
File Edit Format View Help
Action Parameters : {"messageFormat":"JSON","messageArguments":[{"id":12345}]}

2015-07-06 13:40:05.2033|ERROR|WebApi.Helpers.NLogger|Method: GET
URL: http://localhost:55959/v1/Products/Product/all?id=12345
Token: 1608f3d6-484a-4d8f-951e-d2f11f55f13d
Controller : WebApi.Controllers.ProductController
Action : Get
Action Parameters :
Error: No product found for this id

2015-07-06 13:08:56.9234|INFO|WebApi.Helpers.NLogger|JSON
Method: GET
URL: http://localhost:55959/v1/Products/Product/all?id=1
Token: 0f744a60-4625-458a-808d-3bf9f5e3d6b4
Controller : WebApi.Controllers.ProductController
Action : Get
Action Parameters : {"messageFormat":"JSON","messageArguments":[{"id":1}]}

2015-07-06 13:10:29.4581|INFO|WebApi.Helpers.NLogger|JSON
Method: GET
URL: http://localhost:55959/v1/Products/Product/all?id=12345
Token: 0f744a60-4625-458a-808d-3bf9f5e3d6b4
Controller : WebApi.Controllers.ProductController
Action : Get
Action Parameters : {"messageFormat":"JSON","messageArguments":[{"id":12345}]}

2015-07-06 13:10:32.3149|ERROR|WebApi.Helpers.NLogger|Method: GET
URL: http://localhost:55959/v1/Products/Product/all?id=12345
Token: 0f744a60-4625-458a-808d-3bf9f5e3d6b4
Controller : WebApi.Controllers.ProductController
Action : Get
Action Parameters :
Error: No product found for this id.
Error Code: 1001
```

Well, now the log has captured both the event and error of same call on the server, you can see call log details and the error with provided error message in the log with our custom error code, I have only captured error description and error code, but you can add more details in the log as per your application needs.

Update the controller for new Exception Handling

Following is the code for controllers with implementation of custom exception handling and logging –

Product Controller

```
using System.Collections.Generic;

using System.Linq;

using System.Net;

using System.Net.Http;

using System.Web.Http;

using AttributeRouting;

using AttributeRouting.Web.Http;

using BusinessEntities;

using BusinessServices;

using WebApi.ActionFilters;

using WebApi.Filters;

using System;

using WebApi.ErrorHelper;

namespace WebApi.Controllers

{

    [AuthorizationRequired]

    [RoutePrefix("v1/Products/Product")]

    public class ProductController : ApiController
```



```
{  
  
    #region Private variable.  
  
    private readonly IProductServices _productServices;  
  
    #endregion  
  
    #region Public Constructor  
  
    /// <summary>  
    /// Public constructor to initialize product service instance  
    /// </summary>  
    public ProductController(IProductServices productServices)  
    {  
        _productServices = productServices;  
    }  
  
    #endregion  
  
    // GET api/product  
    [GET("allproducts")]  
    [GET("all")]  
    public HttpResponseMessage Get()
```



```

{
    var products = _productServices.GetAllProducts();

    var productEntities = products as List<ProductEntity> ?? products.ToList();

    if (productEntities.Any())

        return Request.CreateResponse(HttpStatusCode.OK, productEntities);

    throw new ApiDataException(1000, "Products not found",
HttpStatusCode.NotFound);
}

```

// GET api/product/5

[GET("productid/{id?}")]

[GET("particularproduct/{id?}")]

[GET("myproduct/{id:range(1, 3)}")]

public HttpResponseMessage Get(int id)

```

{
    if (id != null)
    {
        var product = _productServices.GetProductById(id);

        if (product != null)

            return Request.CreateResponse(HttpStatusCode.OK, product);

        throw new ApiDataException(1001, "No product found for this id.",
HttpStatusCode.NotFound);
    }
}

```



```

    }

    throw new ApiException() { ErrorCode = (int)HttpStatusCode.BadRequest,
        ErrorDescription = "Bad Request..." };
}

```

```
// POST api/product
```

```
[POST("Create")]
```

```
[POST("Register")]
```

```
public int Post([FromBody] ProductEntity productEntity)
```

```

{
    return _productServices.CreateProduct(productEntity);
}

```

```
// PUT api/product/5
```

```
[PUT("Update/productid/{id}")]
```

```
[PUT("Modify/productid/{id}")]
```

```
public bool Put(int id, [FromBody] ProductEntity productEntity)
```

```

{
    if (id > 0)
    {
        return _productServices.UpdateProduct(id, productEntity);
    }

    return false;
}

```



```

    }

    // DELETE api/product/5

    [DELETE("remove/productid/{id}")]
    [DELETE("clear/productid/{id}")]
    [PUT("delete/productid/{id}")]
    public bool Delete(int id)
    {
        if (id != null && id > 0)
        {
            var isSuccess = _productServices.DeleteProduct(id);

            if (isSuccess)
            {
                return isSuccess;
            }

            throw new ApiDataException(1002, "Product is already deleted or not exist in
system.", HttpStatusCode.NoContent );
        }

        throw new ApiException() {ErrorCode = (int) HttpStatusCode.BadRequest,
ErrorDescription = "Bad Request..."};
    }
}
}
}

```


Now you can see, our application is so rich and scalable that none of the exception or transaction can escape logging. Once setup is in place, now you don't have to worry about writing code each time for logging or requests and exceptions, but you can relax and focus on business logic only.



Image credit: <https://pixabay.com/en/kermit-frog-meadow-daisy-concerns-383370/>

Conclusion

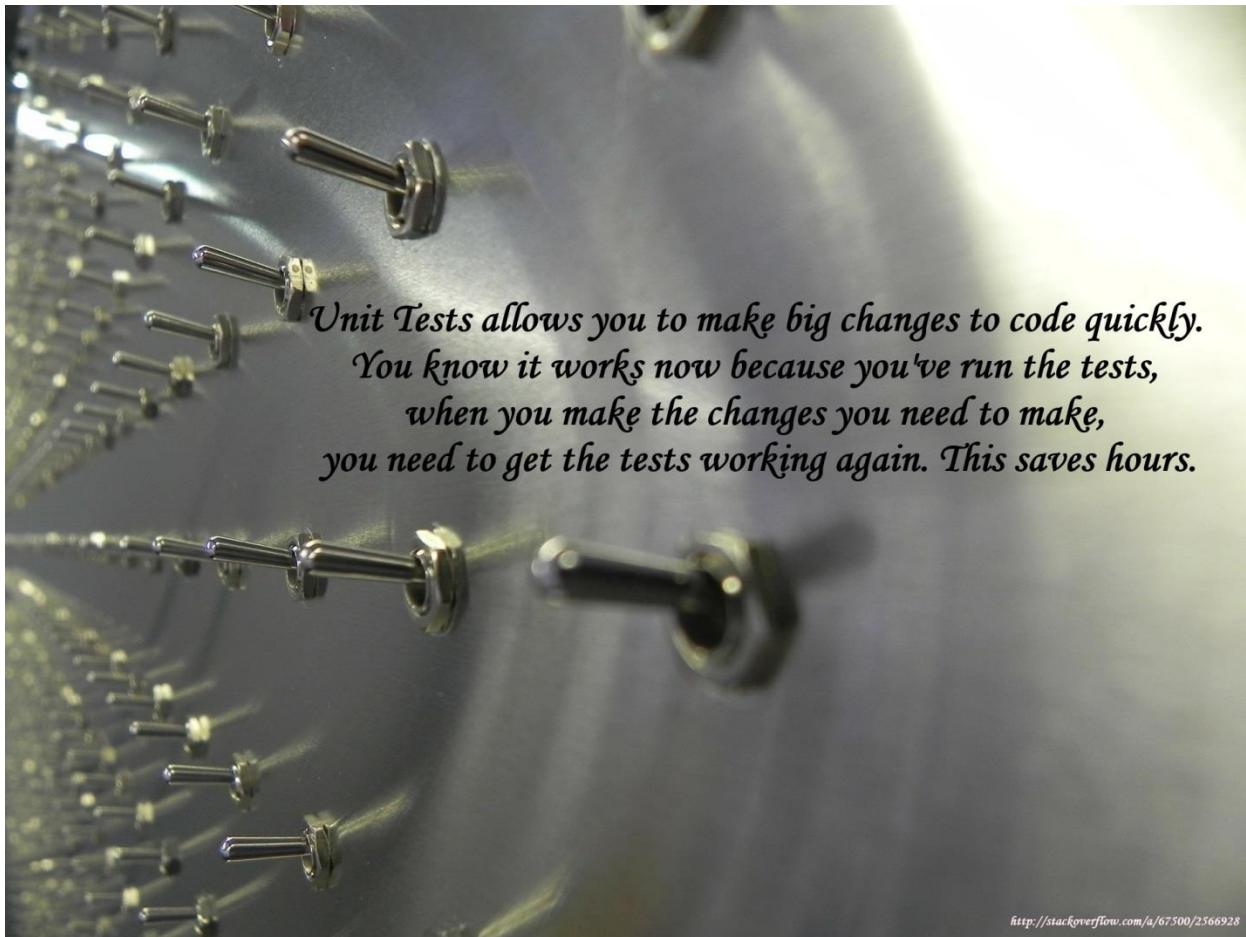
In this article we learnt about how to perform request logging and exception logging in WebAPI. There could be numerous ways in which you can perform these operations but I tried to present this in as simple way as possible. My approach was to take our enterprise level to next level of development, where developers should not always be worried about exception handling and logging. Our solution provides a generic approach of centralizing the operations in one place; all the requests and exceptions are automatically taken care of. In my new articles, I'll try to enhance the application by explaining unit testing in WebAPI and OData in WebAPI. You can download the complete source code of this article with packages from [GitHub](#). Happy coding 😊

Unit testing and Integration Testing in WebAPI using NUnit and Moq framework: Part 1

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.

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](#).



HOME	DOWNLOAD	DOCUMENTATION	WIKI	BLOG	CONTACT US
------	----------	---------------	------	------	------------

Downloads

Current Release: NUnit 3.0.1

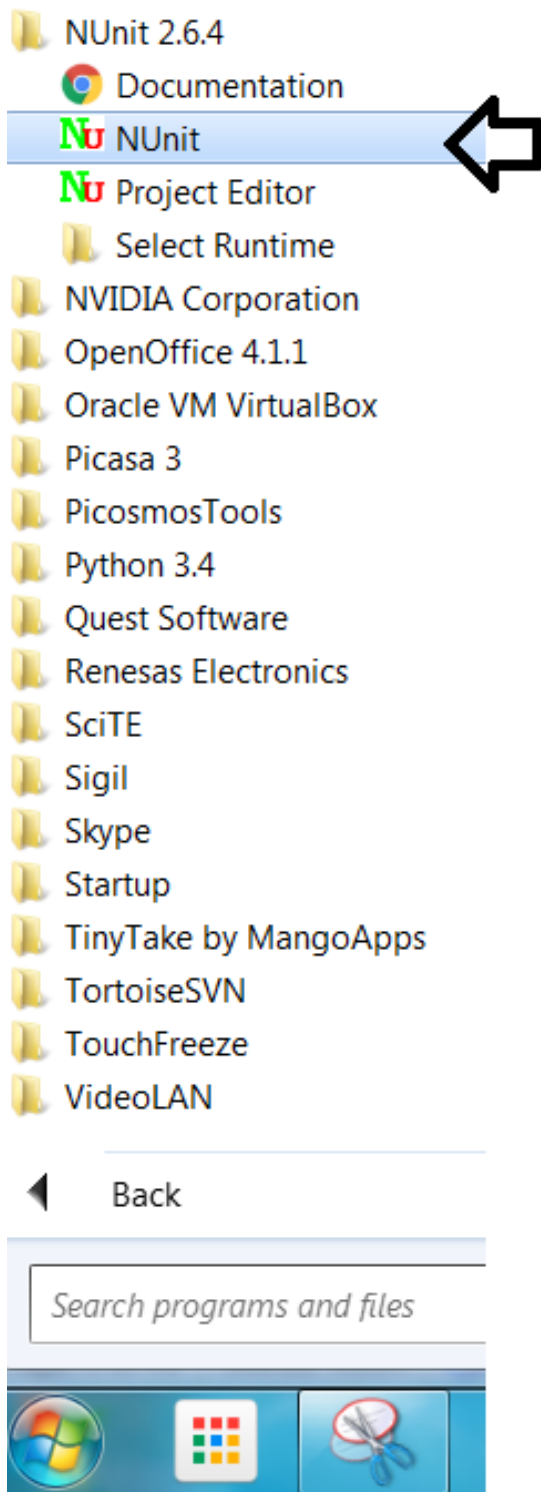
NUnit 3.0.1 - December 1, 2015	
win	NUnit.3.0.1.msi
bin	NUnit-3.0.1.zip
src	NUnit-3.0.1-src.zip
sl	NUnitSL-3.0.1.zip
cf	NUnitCF-3.0.1.zip

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.

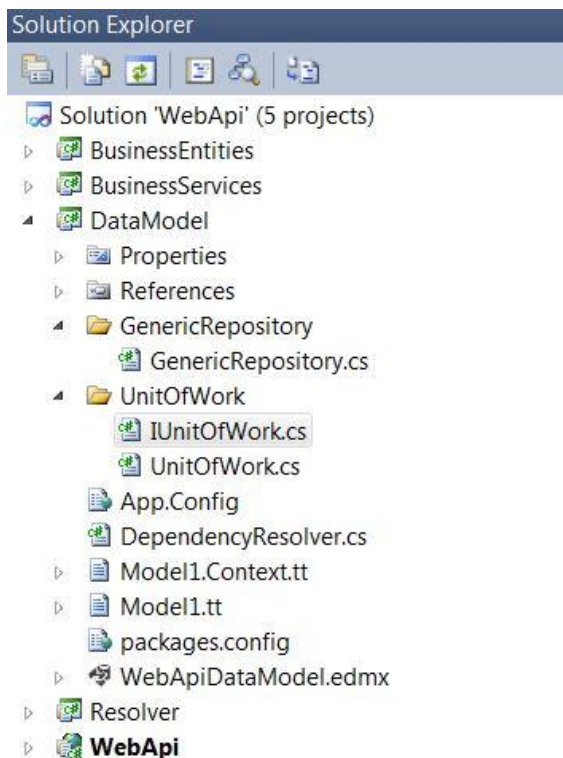
Following is the list taken from [Wikipedia](#) on why to use mock objects,

- 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 IDisposable...
```

```
#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:

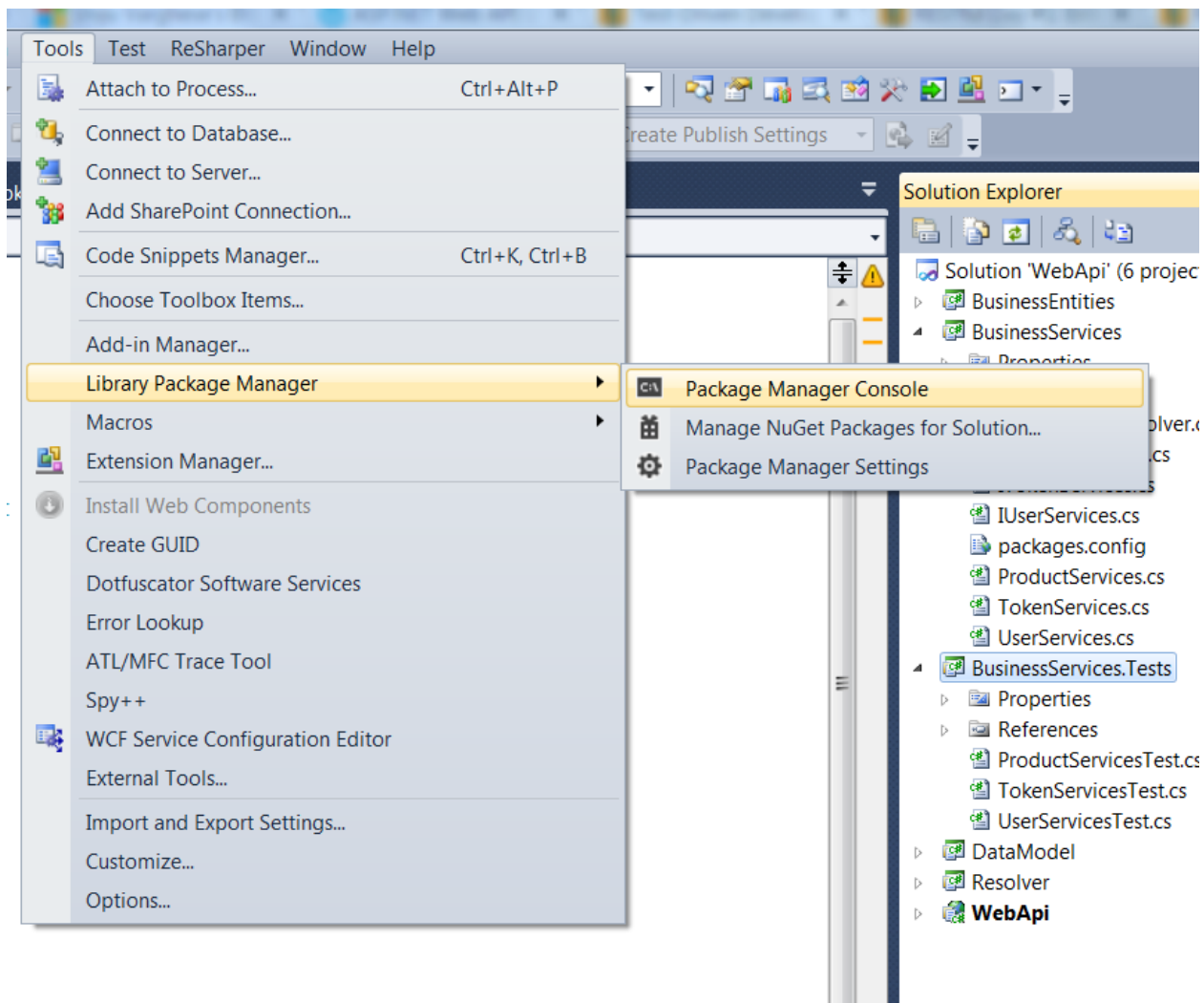

```
#region Private member variables.  
    private readonly IUnitOfWork _unitOfWork;  
#endregion  
|  
#region Public constructor.  
/// <summary>  
/// Public constructor.  
/// </summary>  
public TokenServices(IUnitOfWork unitOfWork)  
{  
    _unitOfWork = unitOfWork;  
}  
#endregion
```

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 some packages before we proceed.



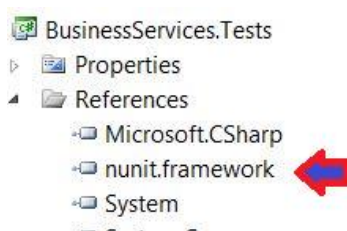
Step 2: Install NUnit package

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


```
Package Manager Console
Package source: nuget.org Default project: BusinessServices.Tests
PM> Install-Package NUnit -Version 2.6.4
Successfully installed 'NUnit 2.6.4'.
Successfully added 'NUnit 2.6.4' to
BusinessServices.Tests.
```

PM>

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



Step 3: Install Moq framework

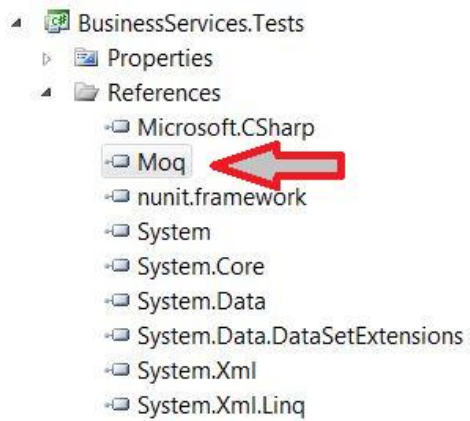
Install the framework on the same project in the similar way as explained in Step 2. Write command “**Install-Package Moq**”. Here we use latest version of Moq

```
Package Manager Console
Package source: nuget.org Default project: BusinessServices.Tests
PM> Install-Package NUnit -Version 2.6.4
Successfully installed 'NUnit 2.6.4'.
Successfully added 'NUnit 2.6.4' to BusinessServices.Tests.

PM> Install-Package Moq
Successfully installed 'Moq 4.2.1510.2205'.
Successfully added 'Moq 4.2.1510.2205' to BusinessServices.Tests.

PM> |
```

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" />

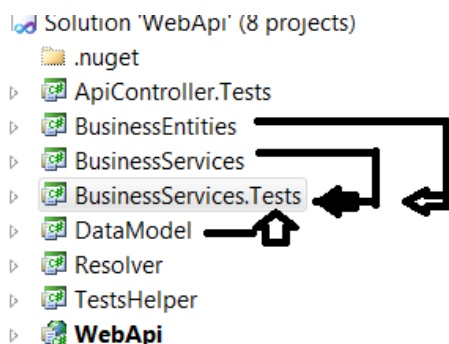
  <package id="Moq" version="4.2.1510.2205" targetFramework="net40" />

  <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
        }
    }
}
```



```
};

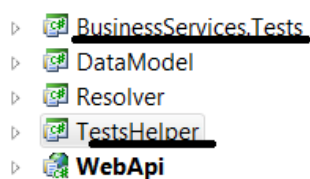
return tokens;
}

/// <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",
```

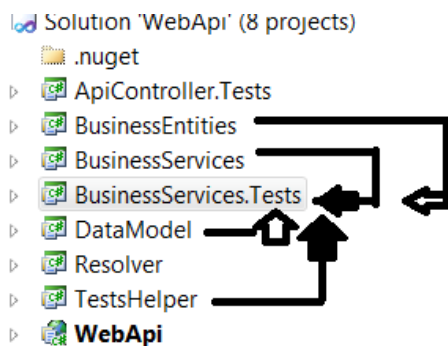
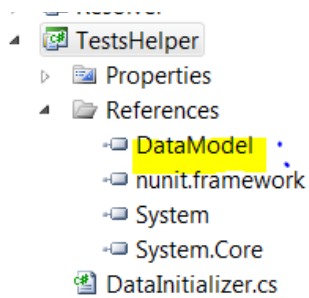


```
Name = "Arsh Mittal",  
  
},  
  
new User()  
  
{  
  
    UserName = "divit",  
  
    Password = "divit",  
  
    Name = "Divit Agarwal",  
  
}  
  
};  
  
  
return users;  
  
}  
  
  
}  
  
}
```

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 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();
```

©2016 Akhil Mittal (www.codet Teddy.com)


```

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);
    }
}

```

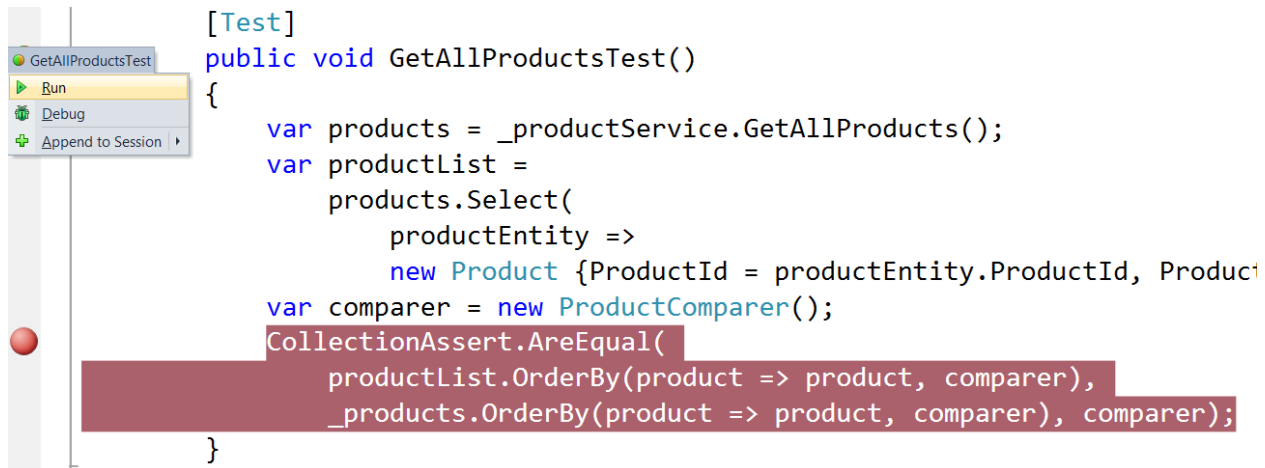
To assert the result we use [CollectionAssert.AreEqual](#) of NUnit where we pass both the lists and comparer.

```

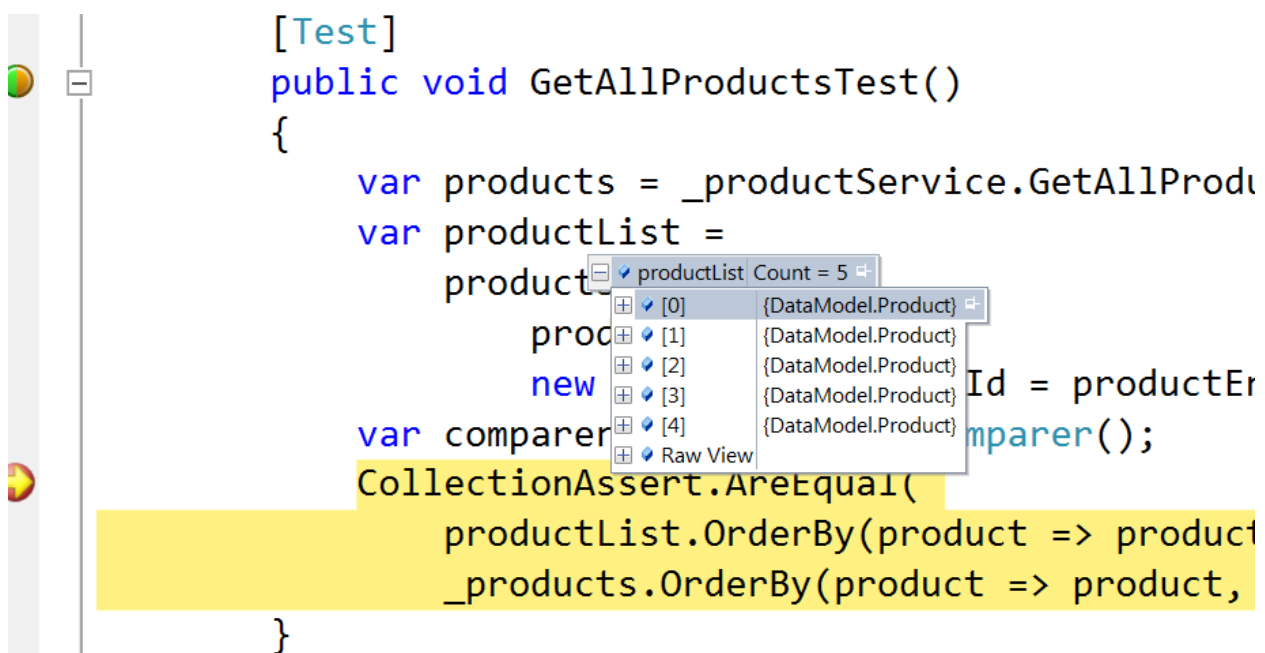
CollectionAssert.AreEqual(
    productList.OrderBy(product => product, comparer),
    _products.OrderBy(product => product, comparer), comparer);

```

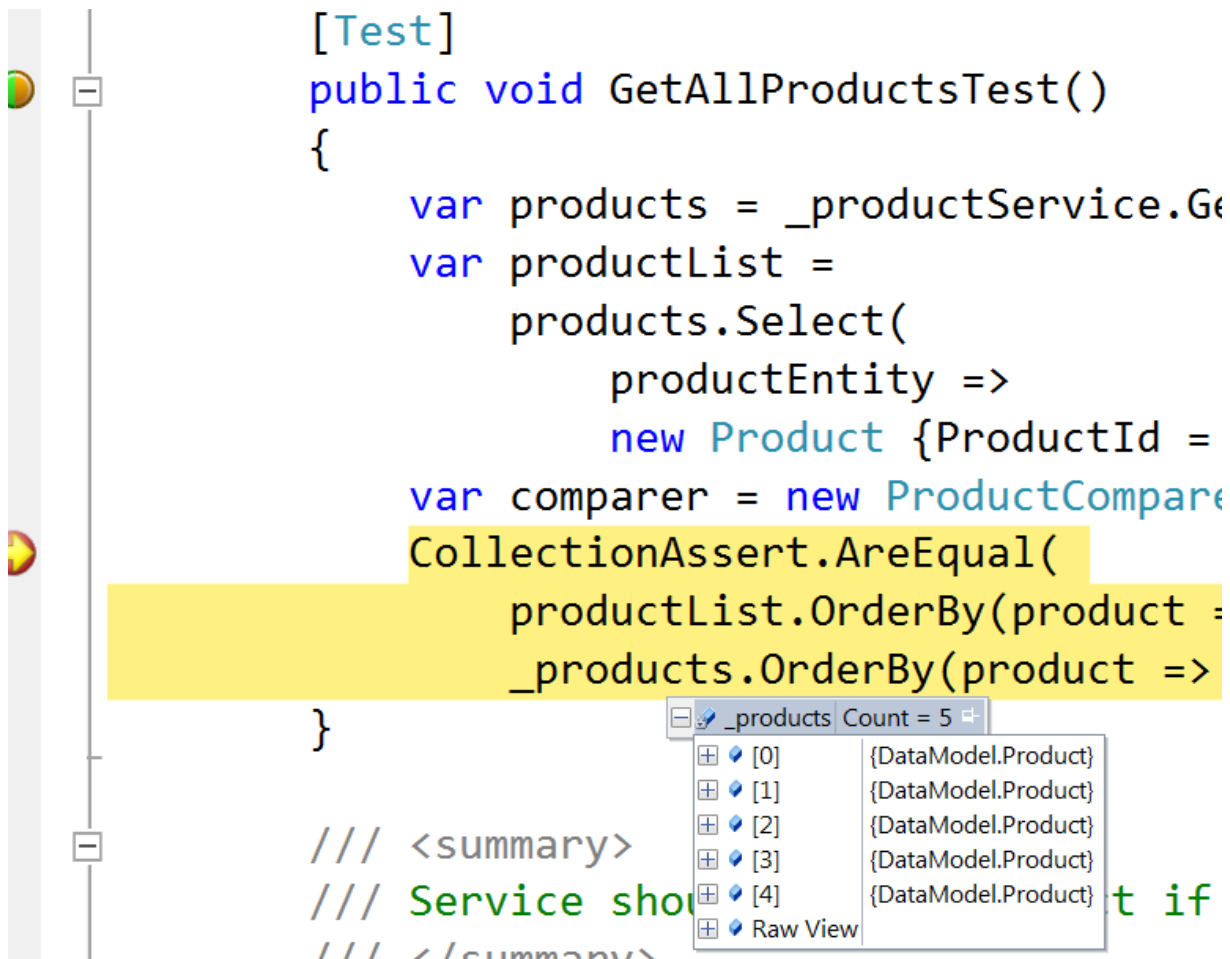

Since I have NUnit plugin in my visual studio provided by Resharper, let me debug the test method to see the actual result of Assert. We'll run all the tests with NUnit UI at the end of the article.



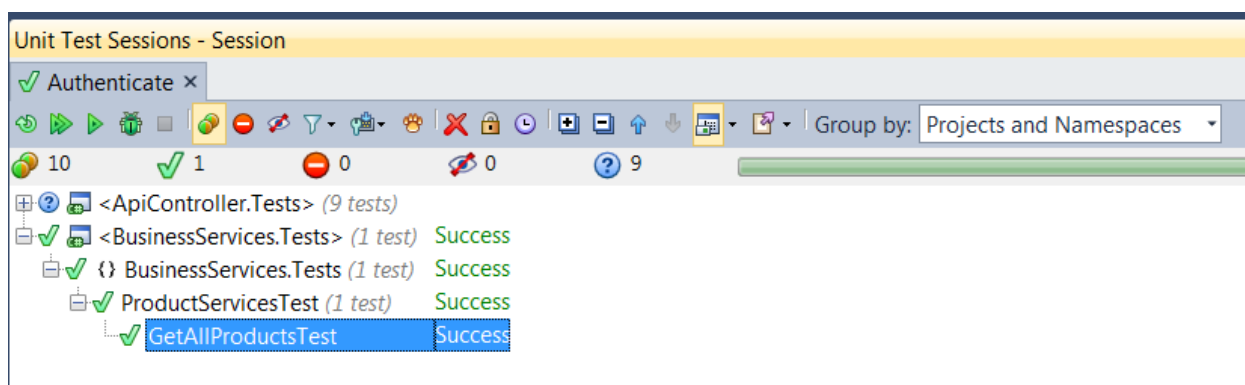
productList ,



_products,



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.

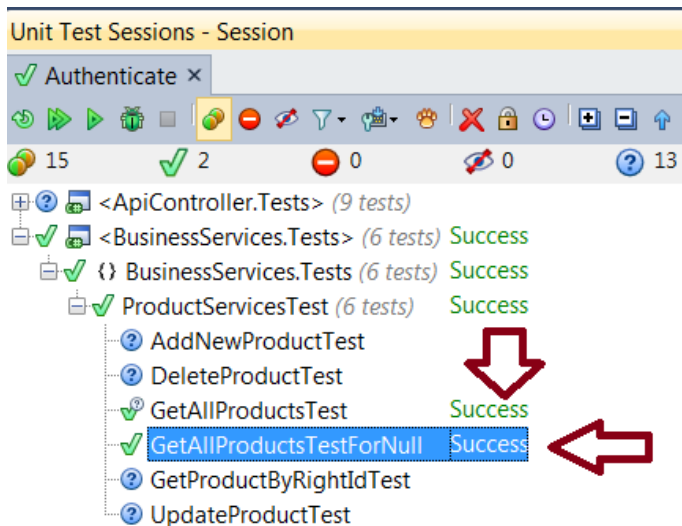
```
/// <summary>
/// Service should return null
/// </summary>
[Test]
public void GetAllProductsTestForNull()
{
    _products.Clear();

    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]))
```

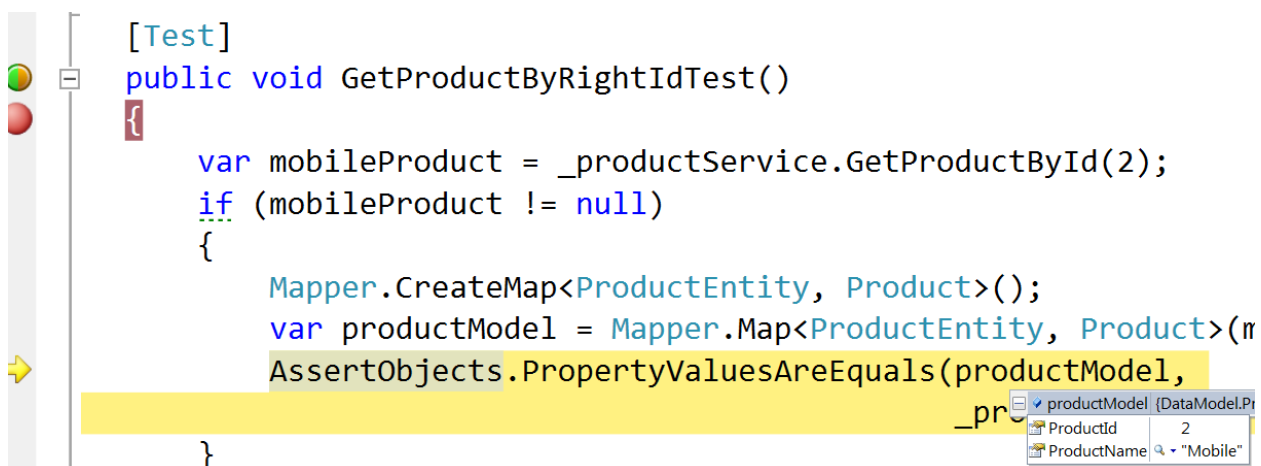
```
Assert.Fail("Property {0}.{1} does not match. Expected IList with element {1} equals to {2}
but was IList with element {1} equals to {3}", property.PropertyType.Name, property.Name,
expectedList[i], actualList[i]);
```

```
}
```

```
}
```

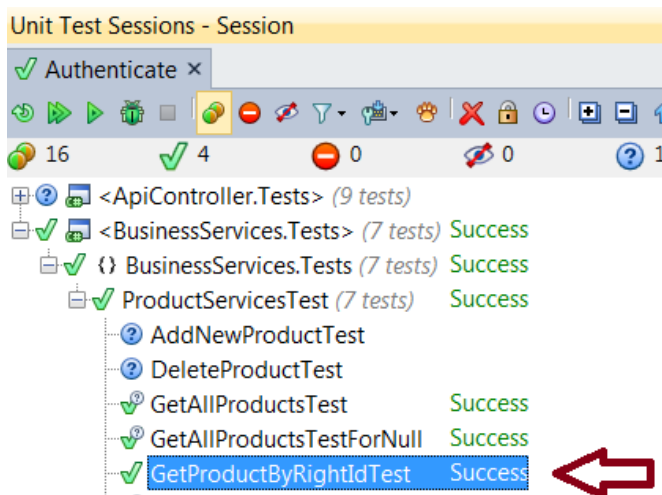
```
}
```

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>(mobileProduct);
        AssertObjects.PropertyValuesAreEquals(productModel, expectedProductModel);
    }
}
```

productModel (DataModel.Pr...	
ProductId	2
ProductName	"Mobile"

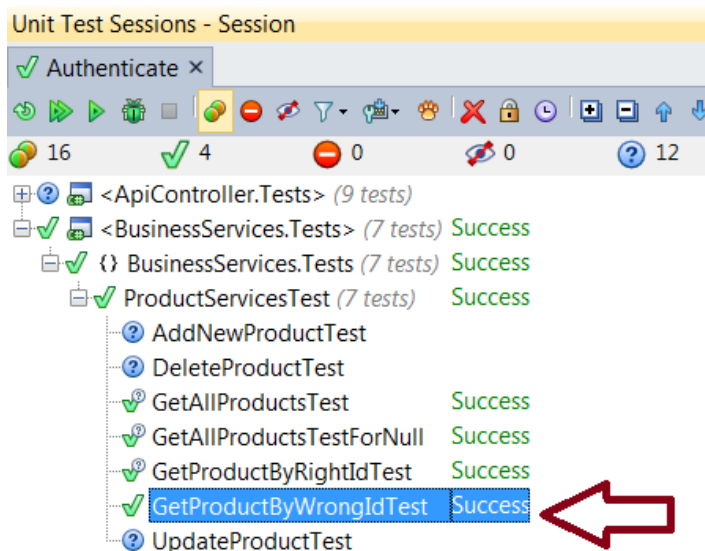


4. GetProductByWrongIdTest ()

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

```

Annotations in the code above:
 - Arrow pointing to `maxProductIDBeforeAdd`: **max id : 5**
 - Arrow pointing to `maxProductIDBeforeAdd + 1`: **new prod id : max +1 = 6**

```

[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]
public void UpdateProductTest()

```

Annotations in the code above:
 - Arrow pointing to `maxProductIDBeforeAdd + 1`: **6**
 - Arrow pointing to `Is.EqualTo(_products.Last().ProductId)`: **6**

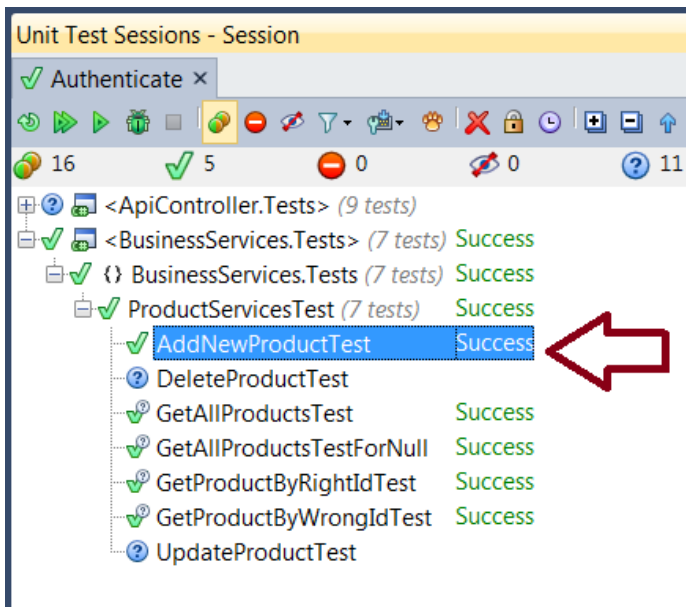
Visual Studio Output Window (Test Results):

Test Name	Result
[0] (DataModel.Product)	Passed
[1] (DataModel.Product)	Passed
[2] (DataModel.Product)	Passed
[3] (DataModel.Product)	Passed
[4] (DataModel.Product)	Passed
[5] (DataModel.Product)	Passed

Product Details:

Property	Value
ProductId	6
ProductName	"Android Phone"

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);

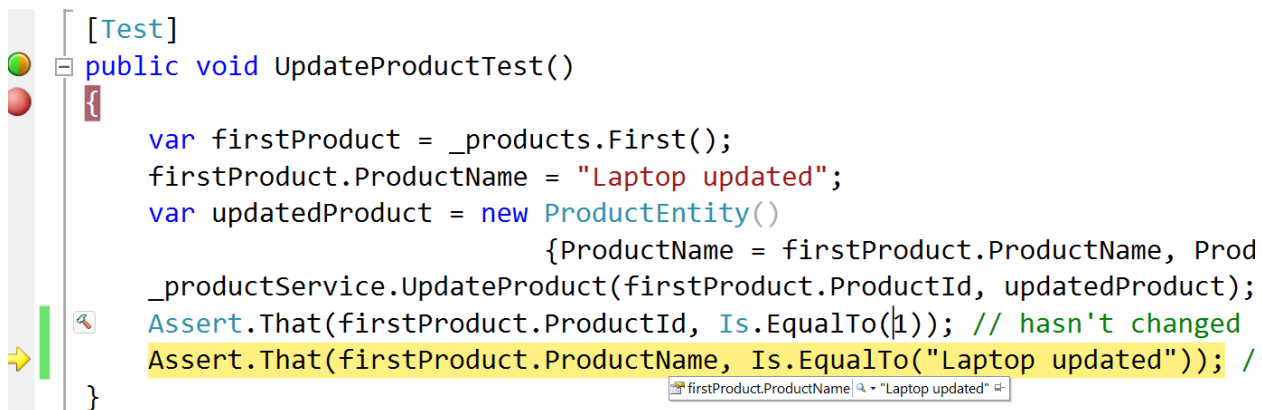
Assert.That(firstProduct.ProductId, Is.EqualTo(1)); // hasn't changed

    Assert.That(firstProduct.ProductName, Is.EqualTo("Laptop updated")); // Product name
changed

}

```

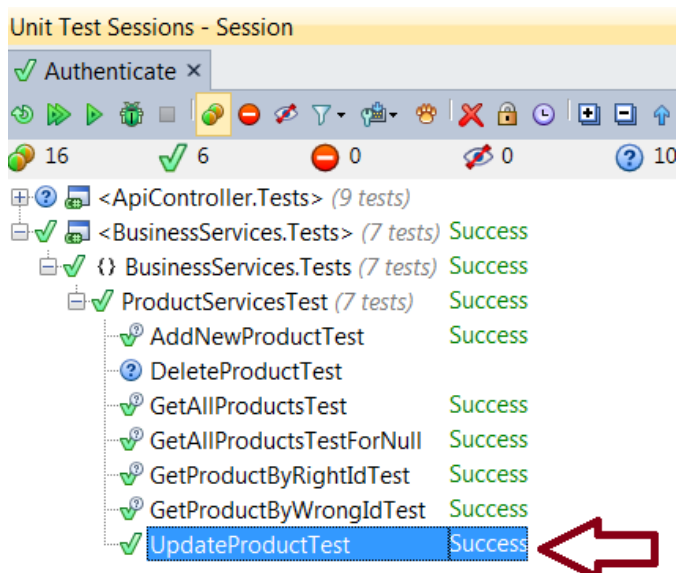
In this test I am trying to update first product from _products list. I have changed the product name to “Laptop Updated” and invoked the UpdateProduct () method of ProductService. I have made two asserts to check the updated product from _products list, one for productId and second for product name. 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, 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")); //
}

```

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);

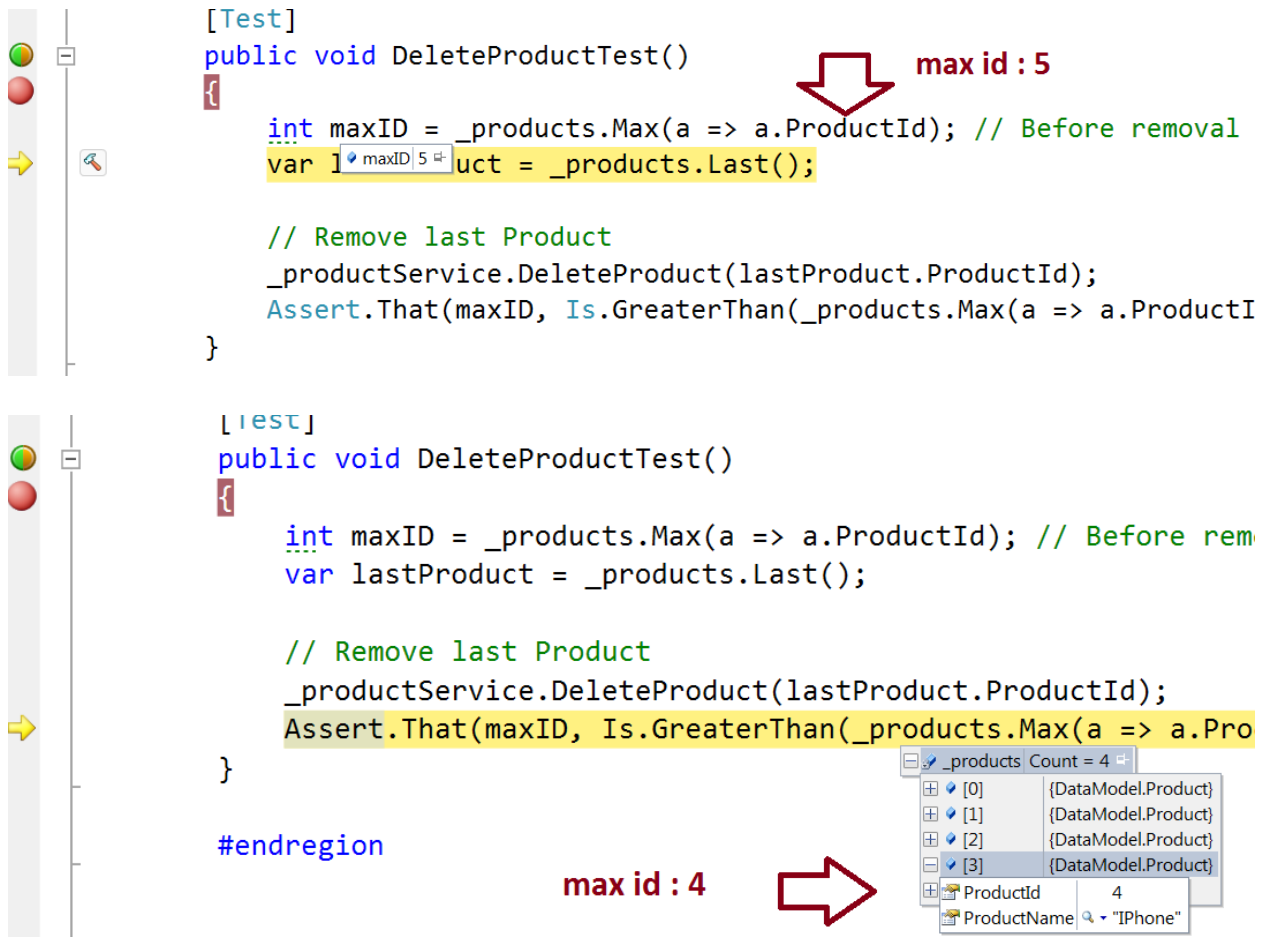
    Assert.That(maxID, Is.GreaterThan(_products.Max(a => a.ProductId))); // Max id
    reduced by 1

```



```
}
```

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.



```
[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)));
}

[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)));
}

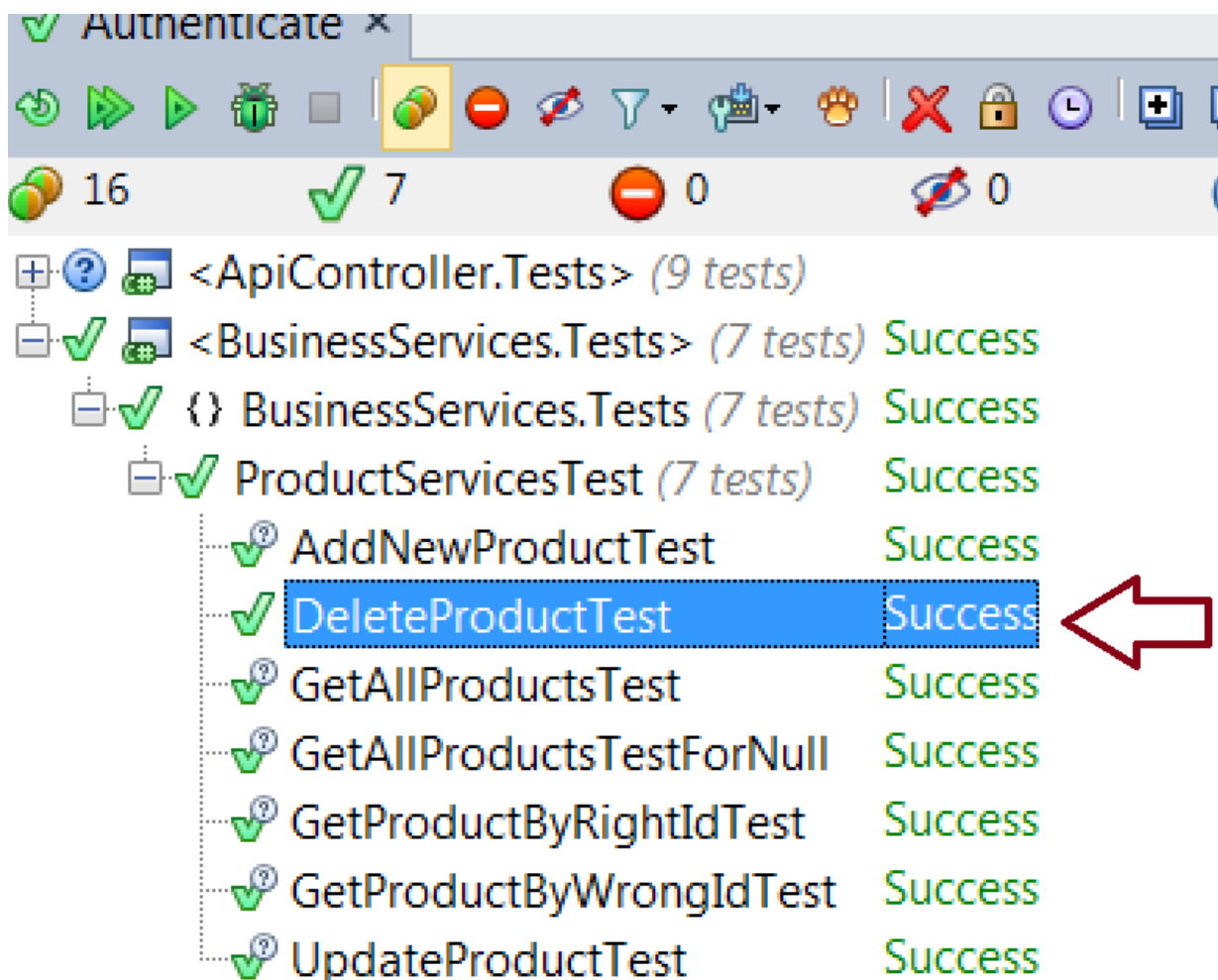
#endregion
```

max id : 5

max id : 4

_products Count = 4	
[0]	{DataModel.Product}
[1]	{DataModel.Product}
[2]	{DataModel.Product}
[3]	{DataModel.Product}
ProductId	4
ProductName	"iPhone"

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 virtuals 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

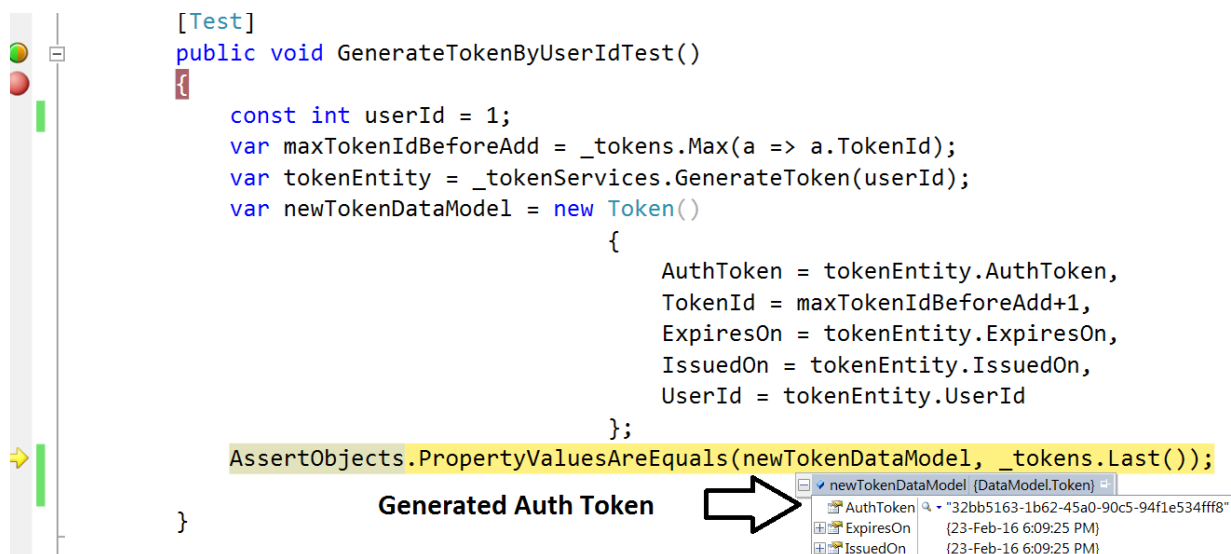
};

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.



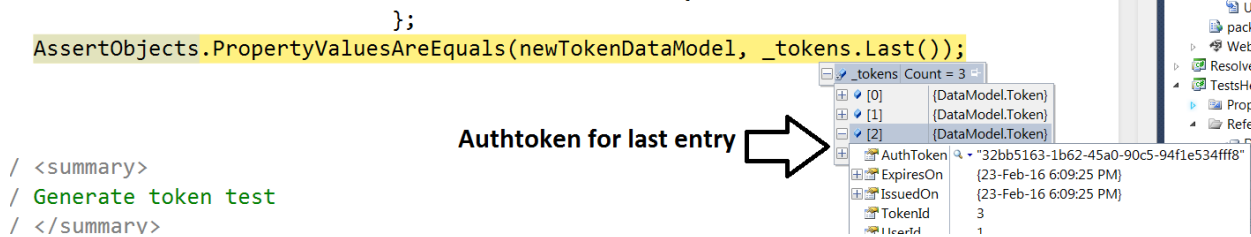
```

[Test]
public void GenerateTokenByIdTest()
{
    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());
}

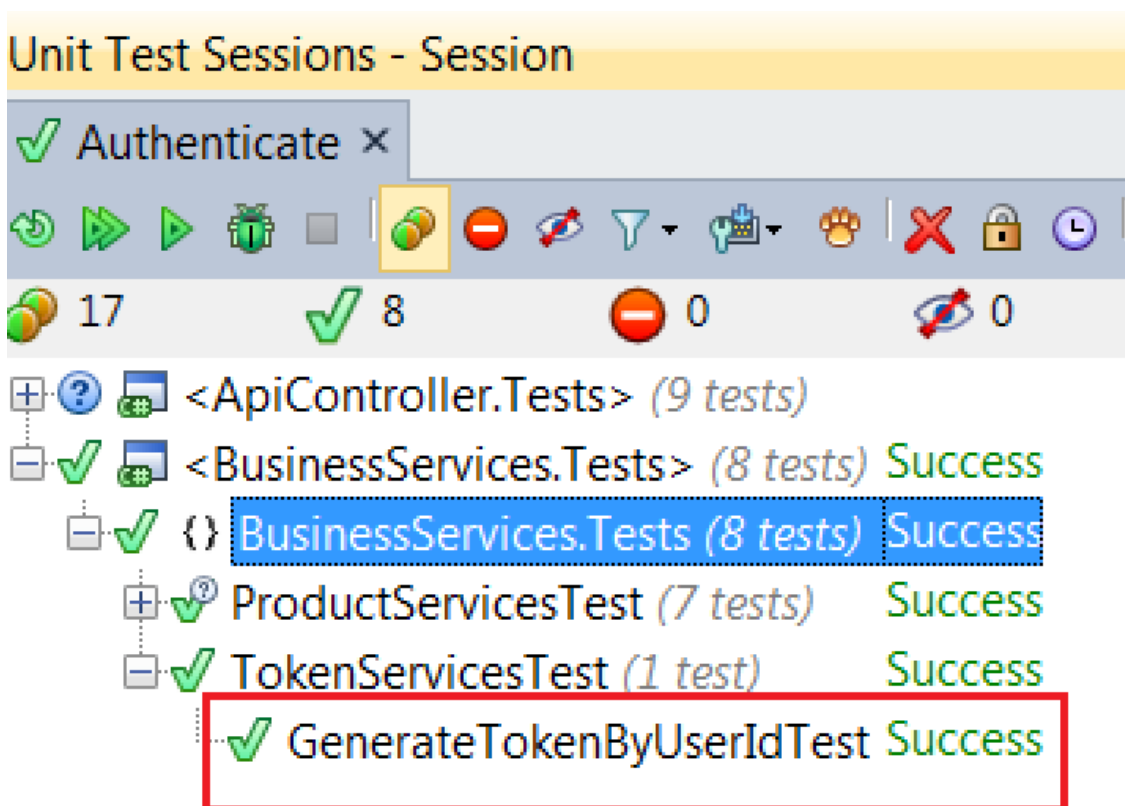
```

Generated Auth Token →

newTokenDataModel (DataModel.Token)	
AuthToken	"32bb5163-1b62-45a0-90c5-94f1e534fff8"
ExpiresOn	(23-Feb-16 6:09:25 PM)
IssuedOn	(23-Feb-16 6:09:25 PM)



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



2. ValidateTokenWithRightAuthToken ()

```
/// <summary>
```

```
/// Validate token test
```

```
/// </summary>
```



```

[Test]

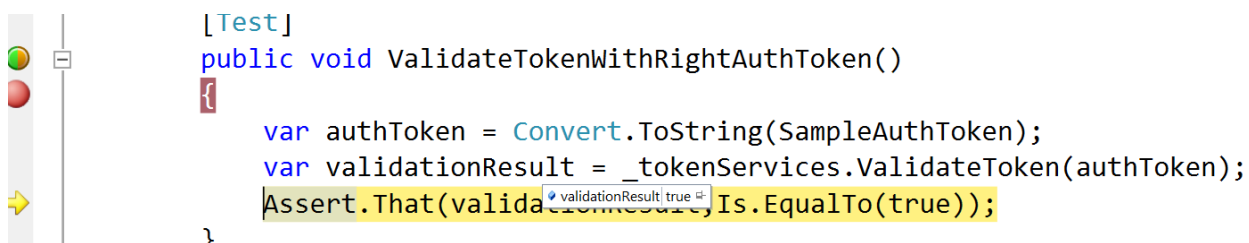
public void ValidateTokenWithRightAuthToken()
{
    var authToken = Convert.ToString(SampleAuthToken);

    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.



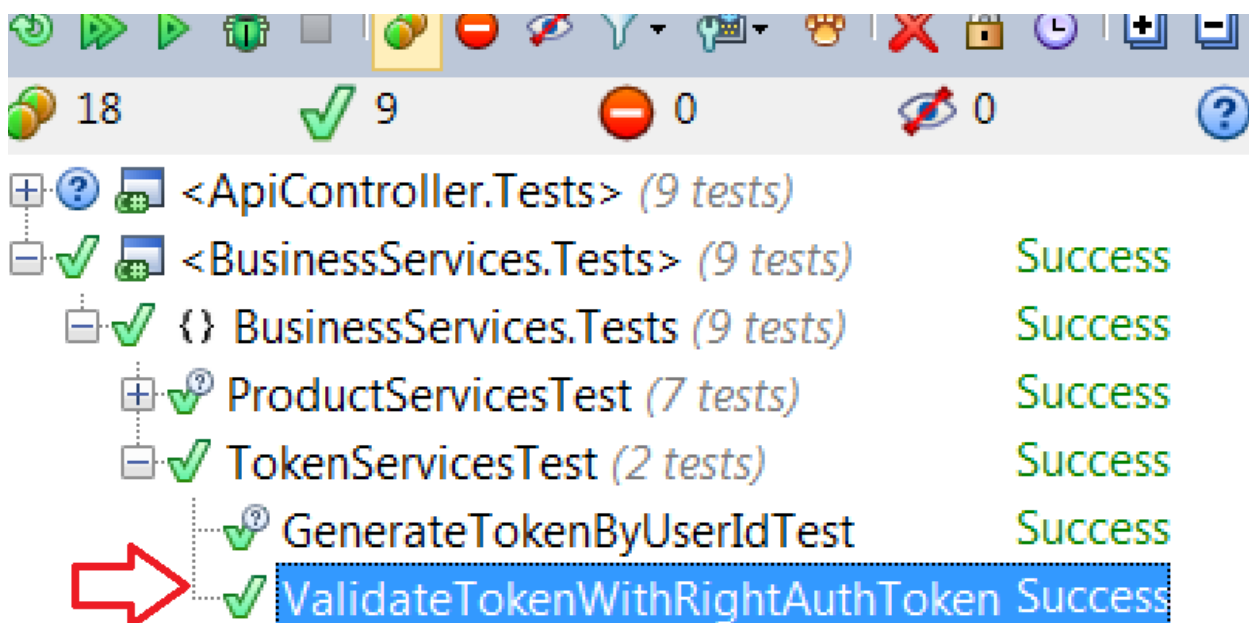
```

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

```

A tooltip is visible over the `validationResult` variable, showing its value as `true`.

Here we get validationResult as true therefore test should pass.



The screenshot shows the Test Explorer in Visual Studio. The test results are as follows:

Test Name	Result
<ApiController.Tests> (9 tests)	
<BusinessServices.Tests> (9 tests)	Success
{ } BusinessServices.Tests (9 tests)	Success
ProductServicesTest (7 tests)	Success
TokenServicesTest (2 tests)	Success
GenerateTokenByUserIdTest	Success
ValidateTokenWithRightAuthToken	Success

A red arrow points to the `ValidateTokenWithRightAuthToken` test, which is highlighted in blue and shows a success status.

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");

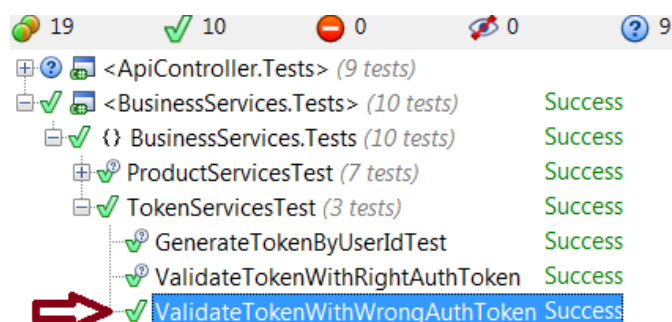
    var validationResult = _tokenServices.ValidateToken(authToken);

    Assert.That(validationResult, Is.EqualTo(false));
}
```

[Test]

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

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

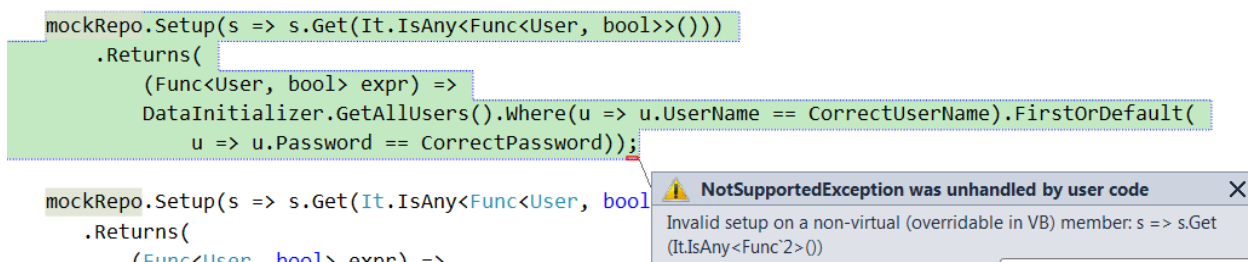


UserService Tests

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,



“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.

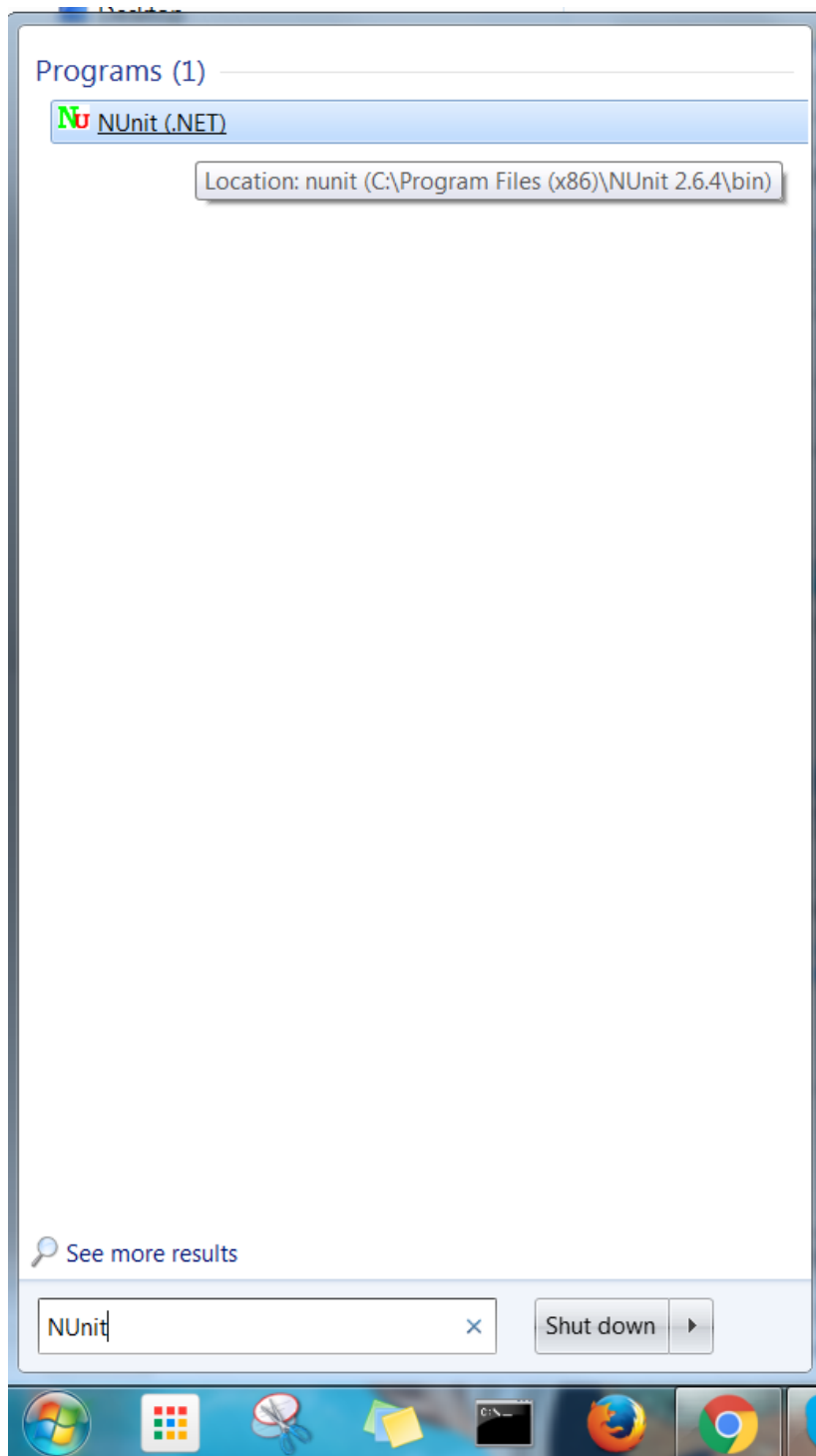
Test through NUnit UI



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

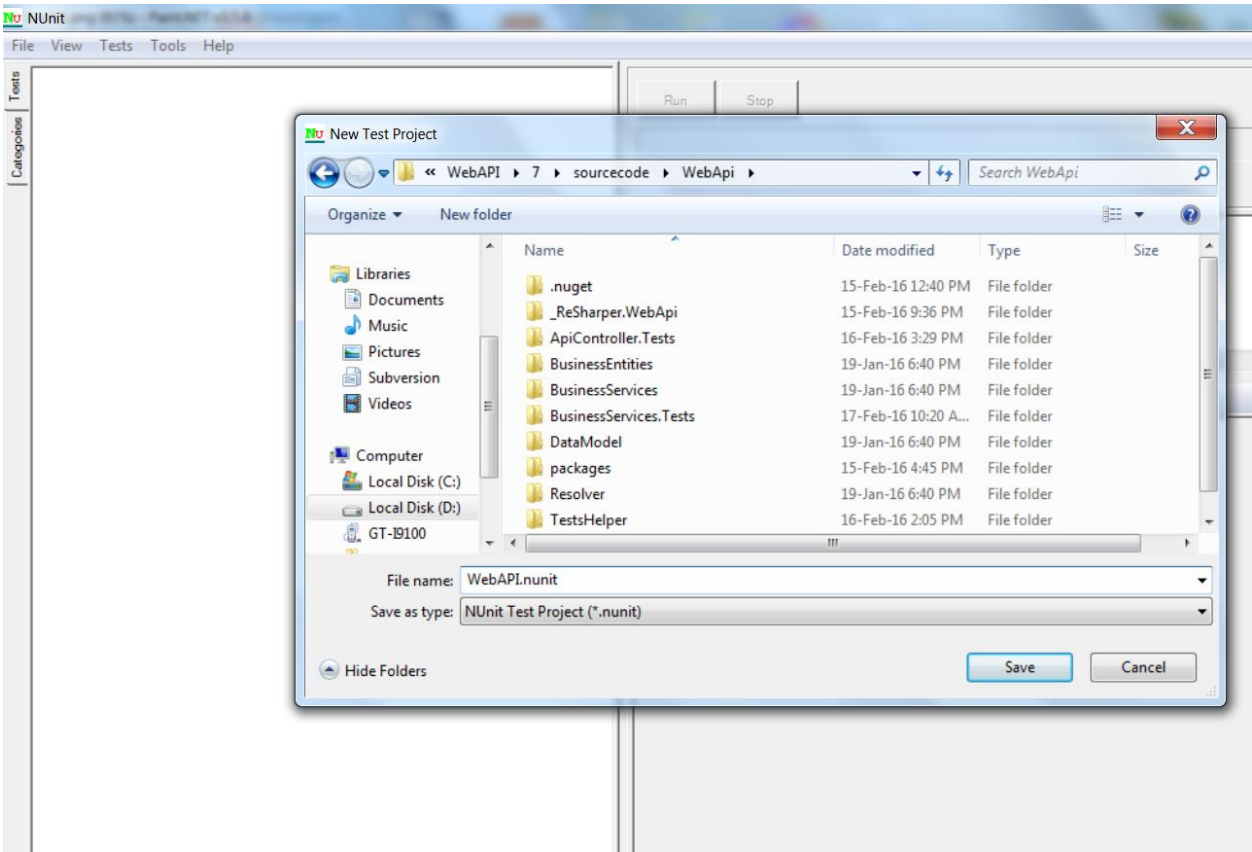
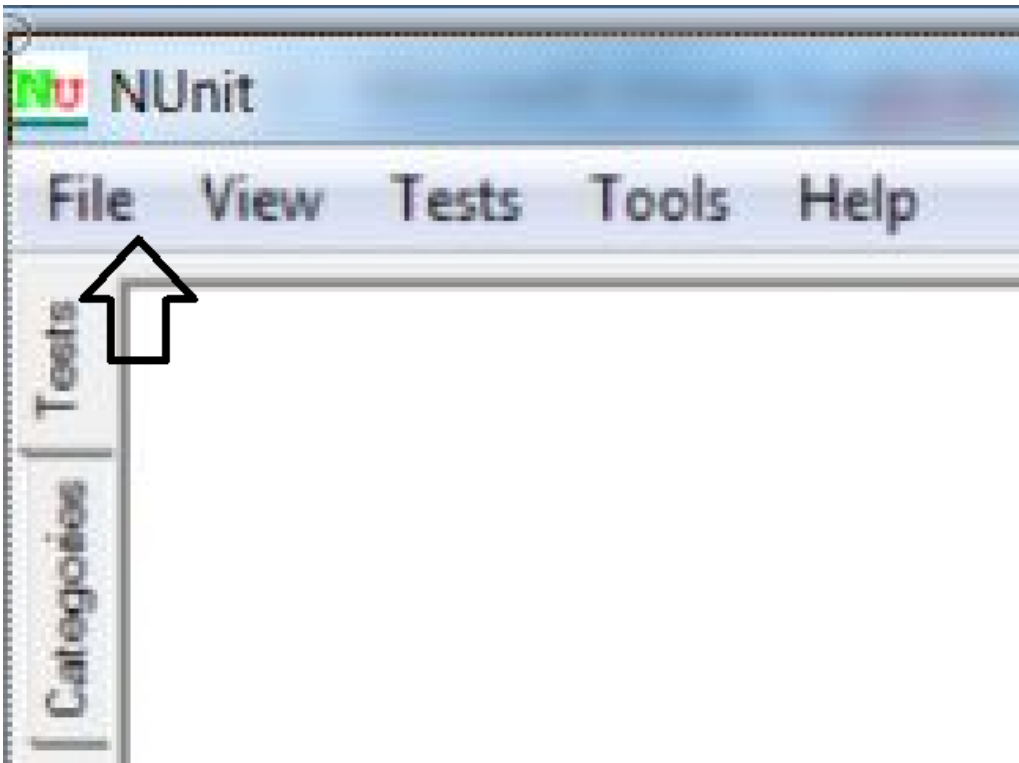
1. Step 1:

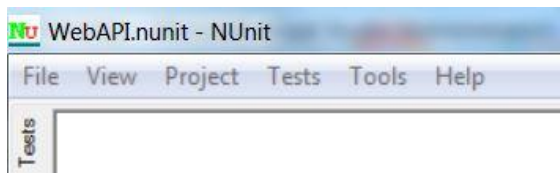
Launch NUnit UI. I have already explained how to install NUnit on the windows machine. Just launch the NUnit interface with its launch icon,



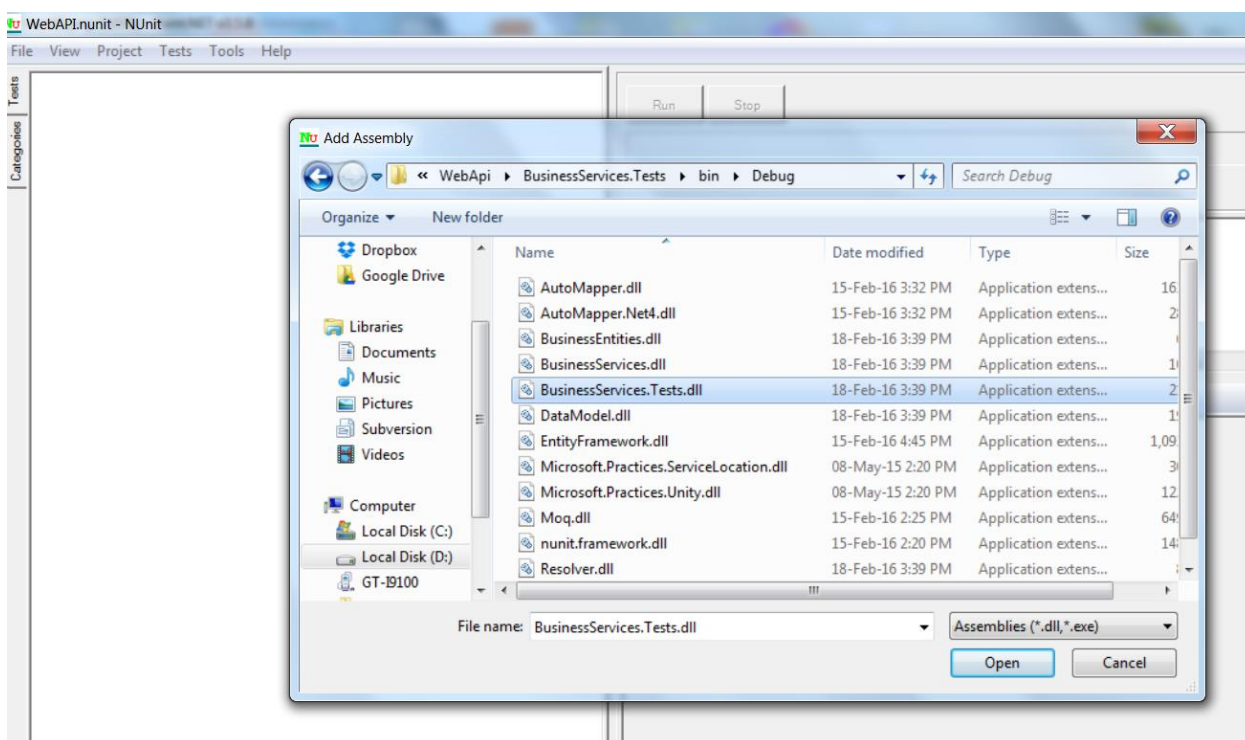
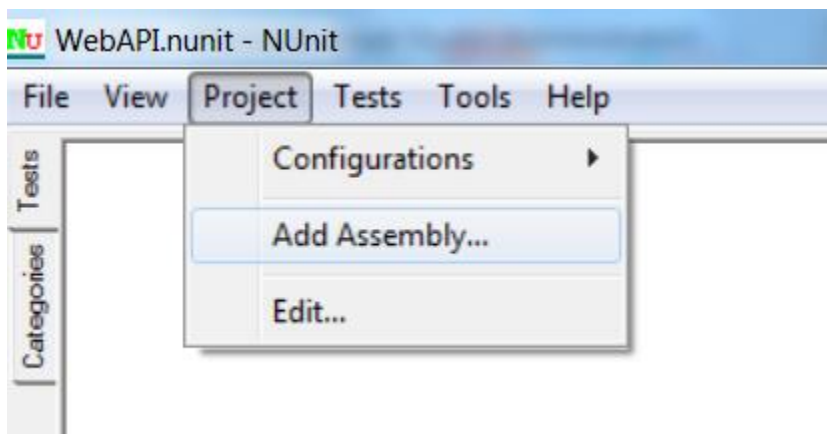
2. Step 2 :

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

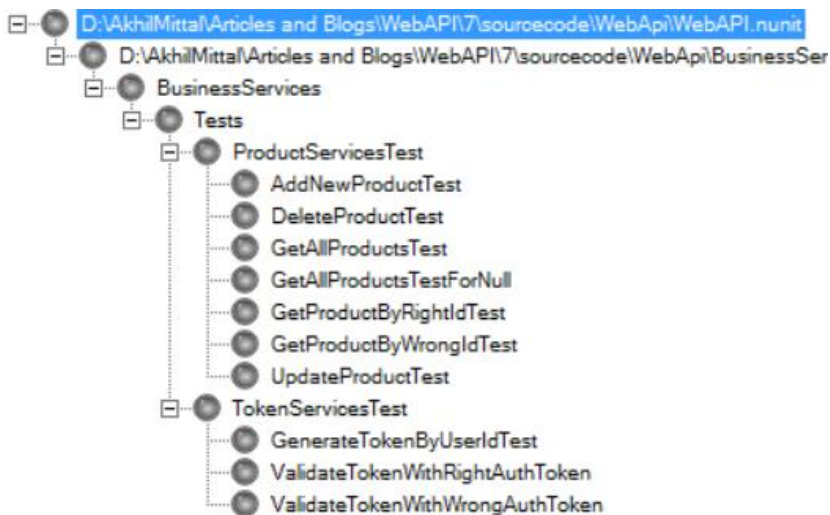




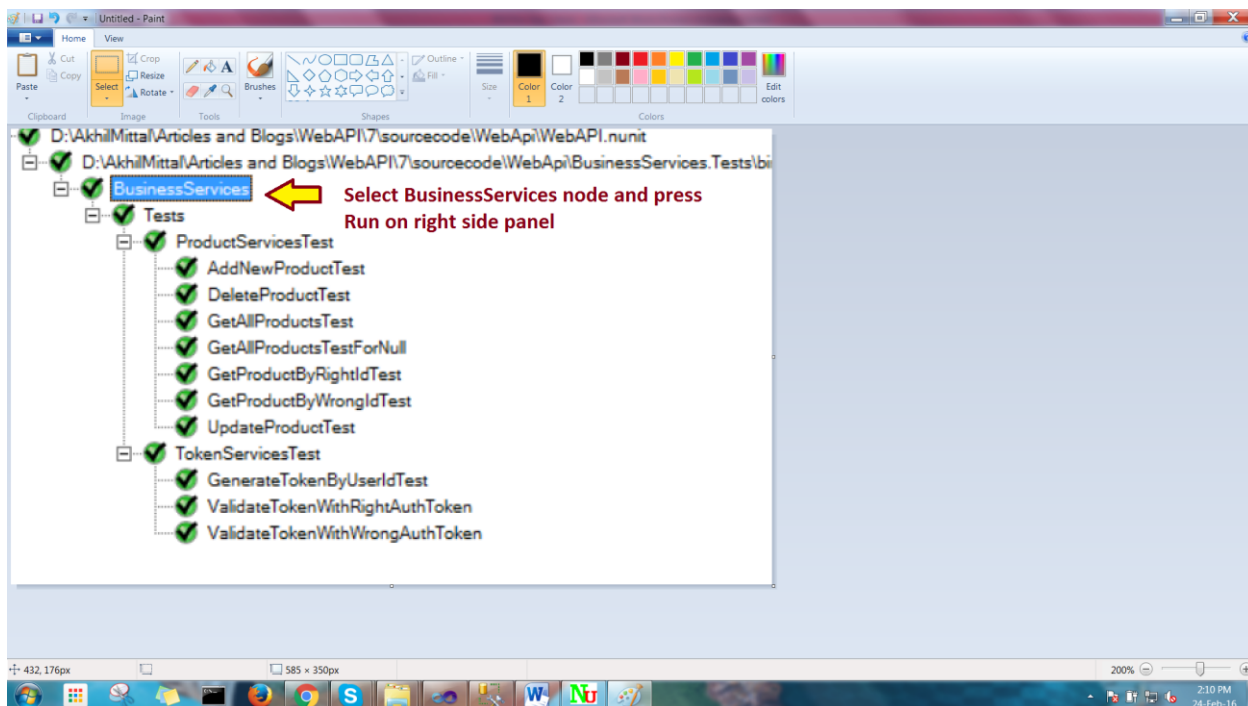
- Step 3: Now, click on Project-> Add Assembly and browse for BusinessServices.Tests.dll (The library created for your unit test project when compiled)



- 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

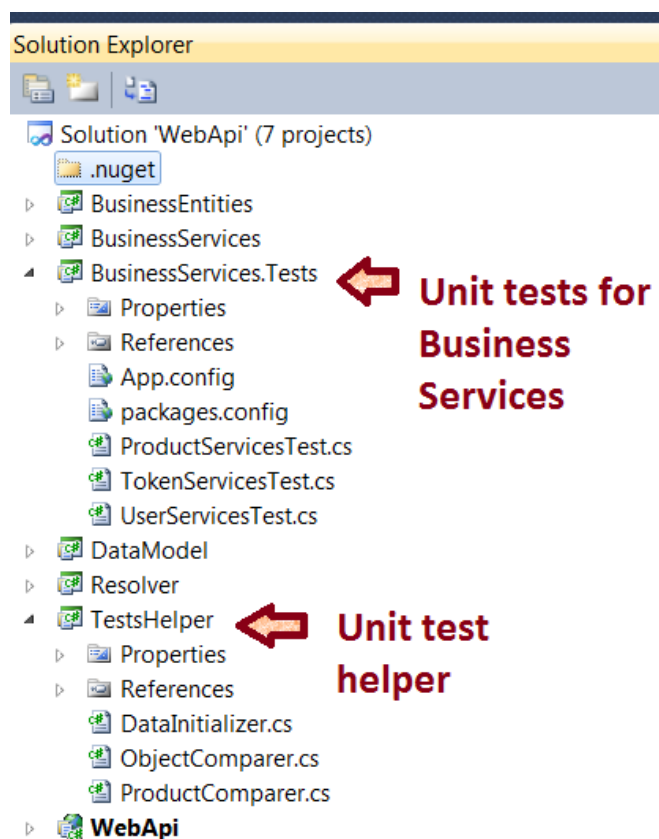
In this article we learnt how to write unit tests for core business logic and primarily on basic CRUD operations. The purpose was to get a basic idea on how unit tests are written and executed. You can add your own flavor to this that helps you in your real time project. My next article which explains 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 😊

Unit testing and Integration Testing in WebAPI Using NUnit and Moq framework: Part 2

In my last article I explained how to write unit tests for business service layer. In this article we'll learn on how to write unit tests for WebAPI controllers i.e. REST's actual endpoints. I'll use [NUnit](#) and [Moq framework](#) to write test cases for controller methods. I have already explained about installing NUnit and configuring unit tests. My last article also covered explaining about NUnit attributes used in writing unit tests.

Setup Solution

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,

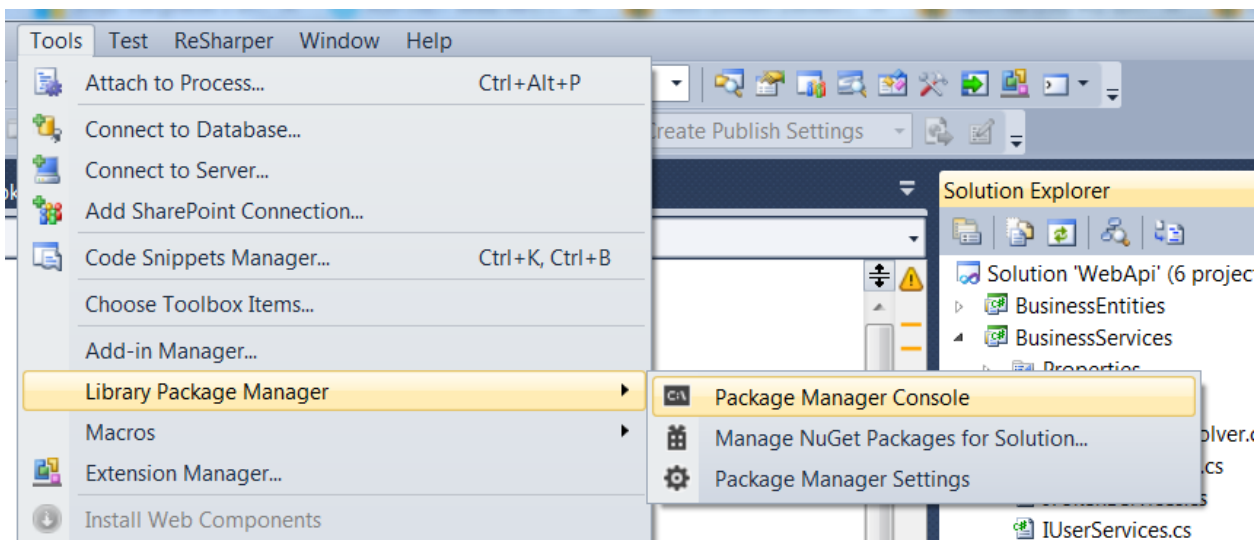


The solution contains the WebAPI application and related projects. There are two newly added projects named BusinessServices.Tests and TestHelper. We'll use TestHelper project and its classes for writing WebAPI unit tests in the same way we used it for writing business services unit tests.

Testing WebAPI

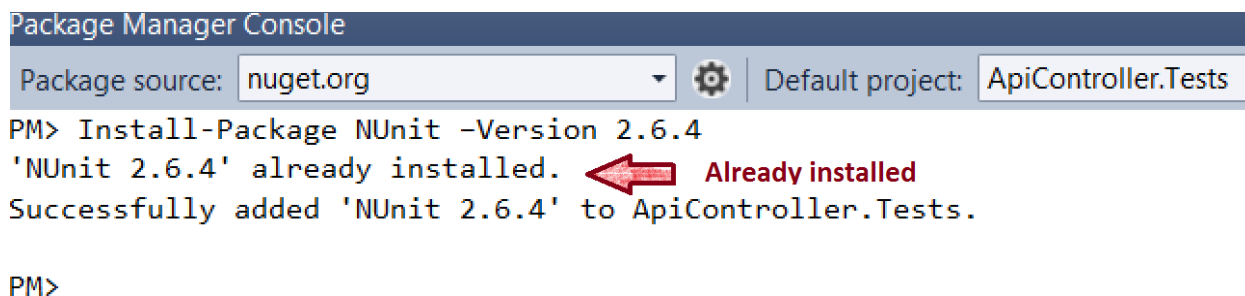
Step 1: Test Project

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



Step 2: Install NUnit package

In package manager console, select ApiController.Tests as default project and write command “**Install-Package NUnit –Version 2.6.4**”. When you run the command, it says NUnit is already installed, that’s because we installed this package for BusinessServices.Tests project, but doing this again for a new project (in our case it is ApiController.Tests) will not install it again but add a reference to nunit framework library and mark an entry in packages.config for ApiController.Tests project.



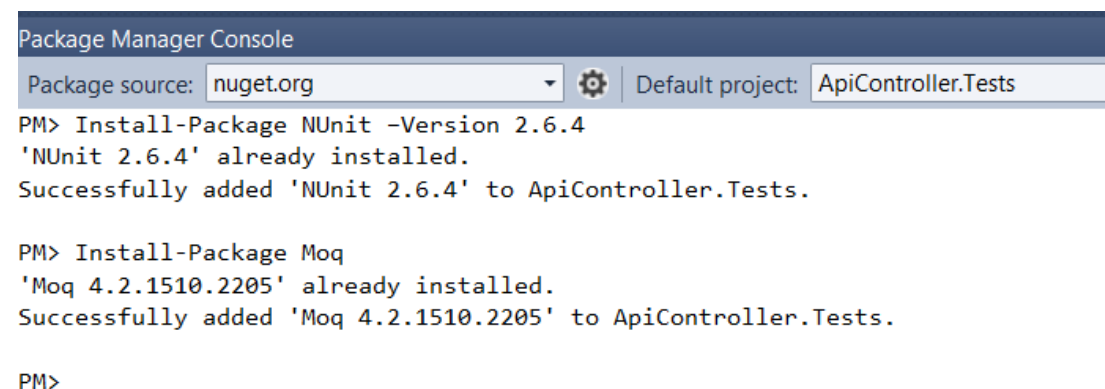
```
Package Manager Console
Package source: nuget.org | Default project: ApiController.Tests
PM> Install-Package NUnit -Version 2.6.4
'NUnit 2.6.4' already installed. ← Already installed
Successfully added 'NUnit 2.6.4' to ApiController.Tests.

PM>
```

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**”.



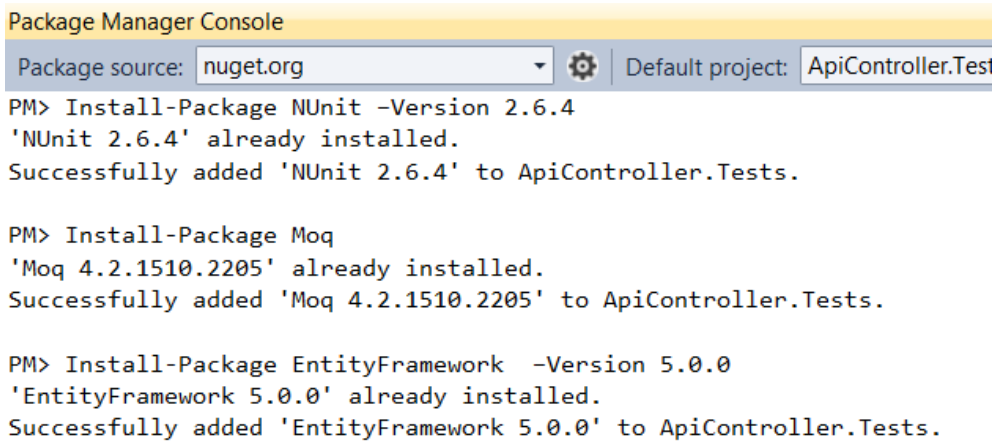
```
Package Manager Console
Package source: nuget.org | Default project: ApiController.Tests
PM> Install-Package NUnit -Version 2.6.4
'NUnit 2.6.4' already installed.
Successfully added 'NUnit 2.6.4' to ApiController.Tests.

PM> Install-Package Moq
'Moq 4.2.1510.2205' already installed.
Successfully added 'Moq 4.2.1510.2205' to ApiController.Tests.

PM>
```

Step 4: Install Entity Framework

Install-Package EntityFramework –Version 5.0.0



```
Package Manager Console
Package source: nuget.org Default project: ApiController.Tests
PM> Install-Package NUnit -Version 2.6.4
'NUnit 2.6.4' already installed.
Successfully added 'NUnit 2.6.4' to ApiController.Tests.

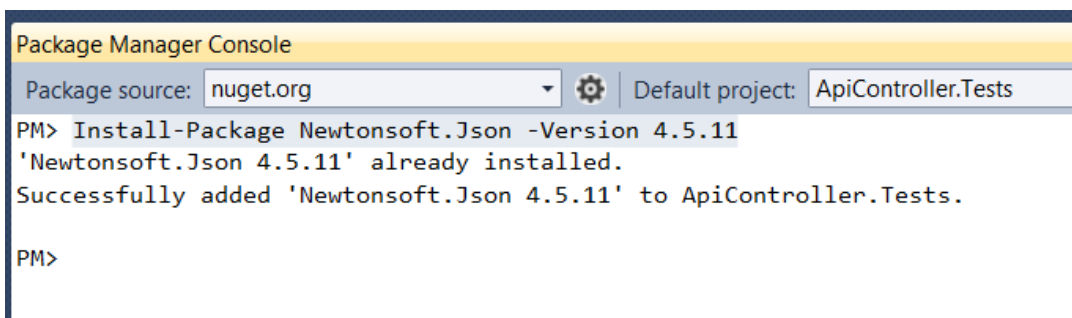
PM> Install-Package Moq
'Moq 4.2.1510.2205' already installed.
Successfully added 'Moq 4.2.1510.2205' to ApiController.Tests.

PM> Install-Package EntityFramework -Version 5.0.0
'EntityFramework 5.0.0' already installed.
Successfully added 'EntityFramework 5.0.0' to ApiController.Tests.
```

Step 5: Newtonsoft.Json

Json.NET is a popular high-performance JSON framework for .NET. We'll use it for serializing/de-serializing request and responses.

Install-Package Newtonsoft.Json -Version 4.5.11



```
Package Manager Console
Package source: nuget.org Default project: ApiController.Tests
PM> Install-Package Newtonsoft.Json -Version 4.5.11
'Newtonsoft.Json 4.5.11' already installed.
Successfully added 'Newtonsoft.Json 4.5.11' to ApiController.Tests.

PM>
```

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

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<packages>
```

```
<package id="EntityFramework" version="5.0.0" targetFramework="net40" />
```

```
<package id="Moq" version="4.2.1510.2205" targetFramework="net40" />
```



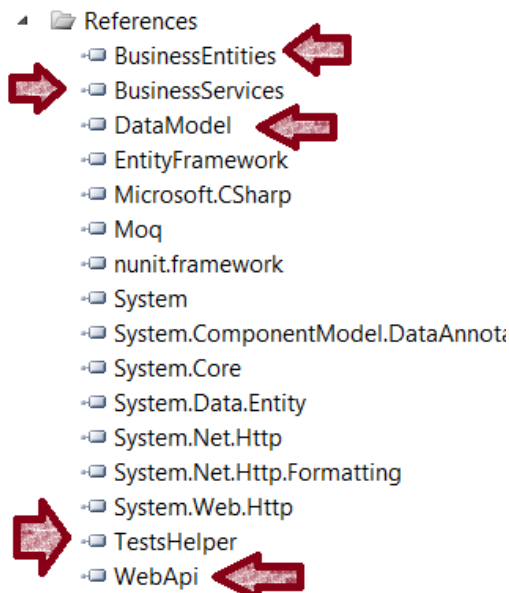
```
<package id="Newtonsoft.Json" version="4.5.11" targetFramework="net40" />
```

```
<package id="NUnit" version="2.6.4" targetFramework="net40" />
```

```
</packages>
```

Step 6: References

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



ProductController 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

Add a new class named `ProductControllerTest.cs` in `ApiController.Tests` project.

Declare variables

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

#region Variables

```
private IProductServices _productService;

private ITokenServices _tokenService;

private IUnitOfWork _unitOfWork;

private List<Product> _products;

private List<Token> _tokens;

private GenericRepository<Product> _productRepository;

private GenericRepository<Token> _tokenRepository;

private WebApiDbEntities _dbEntities;

private HttpClient _client;


private HttpResponseMessage _response;

private string _token;

private const string ServiceBaseURL = "http://localhost:50875/";

#endregion
```


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

Since WebAPI is supposed to return response in `HttpResponse` format so `_response` is declared to store the returned response against which we can assert. `_token` holds the token value after successful authentication. `_client` and `ServiceBaseUrl` may not be required in this article's context, but you can use them to write integration tests that purposely uses actual API URL's and test on actual database.

Write Test Fixture Setup

Write test fixture setup method with `[TestFixtureSetUp]` attribute at the top, this method runs only one time when tests are executed.

```
[TestFixtureSetUp]

public void Setup()
{
    _products = SetUpProducts();
    _tokens = SetUpTokens();
    _dbEntities = new Mock<WebApiDbEntities>().Object;
    _tokenRepository = SetUpTokenRepository();
}
```



```

_productRepository = SetupProductRepository();

var unitOfWork = new Mock<IUnitOfWork>();

unitOfWork.SetupGet(s => s.ProductRepository).Returns(_productRepository);

unitOfWork.SetupGet(s => s.TokenRepository).Returns(_tokenRepository);

_unitOfWork = unitOfWork.Object;

_productService = new ProductServices(_unitOfWork);

_tokenService = new TokenServices(_unitOfWork);

_client = new HttpClient { BaseAddress = new Uri(ServiceBaseURL) };

var tokenEntity = _tokenService.GenerateToken(1);

_token = tokenEntity.AuthToken;

_client.DefaultRequestHeaders.Add("Token", _token);

}

```

The purpose of this method is similar to that of method we wrote for business services. here SetupProducts() will fetch the list of dummy products and SetupTokens() will get the list of dummy tokens. We try to setup mock for Token and Product repository as well, and after that mock UnitOfWork and set it up against already mocked token and product repository. _productService and _tokenService are the instances of ProductService and TokenService respectively, both initialized with mocked Unit of Work.

Following is the line of code that I would like to explain more,

```

var tokenEntity = _tokenService.GenerateToken(1);

_token = tokenEntity.AuthToken;

_client.DefaultRequestHeaders.Add("Token", _token);

```


In the above code we are initializing the `_client` i.e. `HttpClient` with `Token` value in request header. We are doing this because, if you remember about the security (Authentication and Authorization) we implemented in `Product Controller`, which says no request will be entertained unless it is authorized i.e. contains an authentication token in its header. So here we generate the token via `TokenService`'s `GenerateToken()` method, passing a default used id as "1", and use that token for authorization. We require this only to perform integration testing as for unit tests we would directly be calling controller methods from our unit tests methods, but for actual integration tests you'll have to mock all pre-conditions before calling an API endpoint.

`SetUpProducts()`:

```
private static List<Product> SetUpProducts()
{
    var prodId = new int();

    var products = DataInitializer.GetAllProducts();

    foreach (Product prod in products)
        prod.ProductId = ++prodId;

    return products;
}
```

`SetUpTokens()`:

```
private static List<Token> SetUpTokens()
{
    var tokId = new int();
```



```
var tokens = DataInitializer.GetAllTokens();

foreach (Token tok in tokens)

    tok.TokenId = ++tokId;

return tokens;
}
```

Write Test Fixture Tear Down

Unlike [TestFixtureTearDown] tear down is used to de-allocate or dispose the objects.

Following is the code for teardown.

```
[TestFixtureTearDown]

public void DisposeAllObjects()

{

    _tokenService = null;

    _productService = null;

    _unitOfWork = null;

    _tokenRepository = null;

    _productRepository = null;

    _tokens = null;

    _products = null;
}
```



```
    if (_response != null)
        _response.Dispose();

    if (_client != null)
        _client.Dispose();
}
```

Write Test Setup

In this case Setup is only required if you write integration test. So you can choose to omit this.

```
[SetUp]

public void ReInitializeTest()
{
    _client = new HttpClient { BaseAddress = new Uri(ServiceBaseURL) };
    _client.DefaultRequestHeaders.Add("Token", _token);
}
```

Write Test Tear down

Test [TearDown] is invoked after every test execution is complete.

```
[TearDown]

public void DisposeTest()
```



```

{
    if (_response != null)
        _response.Dispose();

    if (_client != null)
        _client.Dispose();
}

```

Mocking Repository

I have created a method `SetUpProductRepository()` to mock Product Repository and assign it to `_productrepository` in `ReInitializeTest()` method and `SetUpTokenRepository()` to mock TokenRepository and assign that to `_tokenRepository` in `ReInitializeTest()` method.

`SetUpProductRepository()`:

```

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;
```

```
}
```

```
SetUpTokenRepository():
```

```
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.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);
});

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

Unit Tests



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

8. GetAllProductsTest ()

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

[Test]

```
public void GetAllProductsTest()
{
    var productController = new ProductController(_productService)
    {
        Request = new HttpRequestMessage
        {
            Method = HttpMethod.Get,
            RequestUri = new Uri(ServiceBaseURL + "v1/Products/Product/all")
        }
    };

    productController.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey, new
    HttpConfiguration());

    _response = productController.Get();
```



```

var responseResult =
JsonConvert.DeserializeObject<List<Product>>(_response.Content.ReadAsStringAsync().Result);

Assert.AreEqual(_response.StatusCode, HttpStatusCode.OK);

Assert.AreEqual(responseResult.Any(), true);

var comparer = new ProductComparer();

CollectionAssert.AreEqual(
responseResult.OrderBy(product => product, comparer),
_products.OrderBy(product => product, comparer), comparer);
}

```

Let me explain the code step by step. We start code by creating an instance of ProductController and initialize the Request property of controller with new request message stating calling http method as GET and initialize the RequestUri with base hosted service URL and appended with actual end point of the method. Initializing RequestUri is not necessary in this case but will help you if you test actual service end point. In this case we are not testing actual endpoint but the direct controller method.

`HttpPropertyKeys.HttpConfigurationKey, new HttpConfiguration()` line adds default httpconfiguration to HttpConfigurationKey necessary for controller instance instantiation.

`_response = productController.Get();` line calls the Get() method of controller that fetches all the products from dummy _products list. Since the return type of the method was an http response message, we need to parse it to get the JSON result sent from the method. All the transactions from API's should ideally happen in form of JSON or XML only. This helps client to understand the response and its result set. We de-serialize the object we got from _response using Newtonsoft library into a list of products. That means the JSON response is converted to List<Product> for better accessibility and comparison. Once done with JSON to List object conversion, I have put three asserts to test the result.

`Assert.AreEqual(_response.StatusCode, HttpStatusCode.OK);` line checks the http status code of the response, the expected is HttpStatusCode.OK.

Second assert i.e. `Assert.AreEqual(responseResult.Any(), true);` checks that we have got the items in the list or not. Third assert is the actual confirmation assert for our test that compares each product from the actual product list to the returned product list.



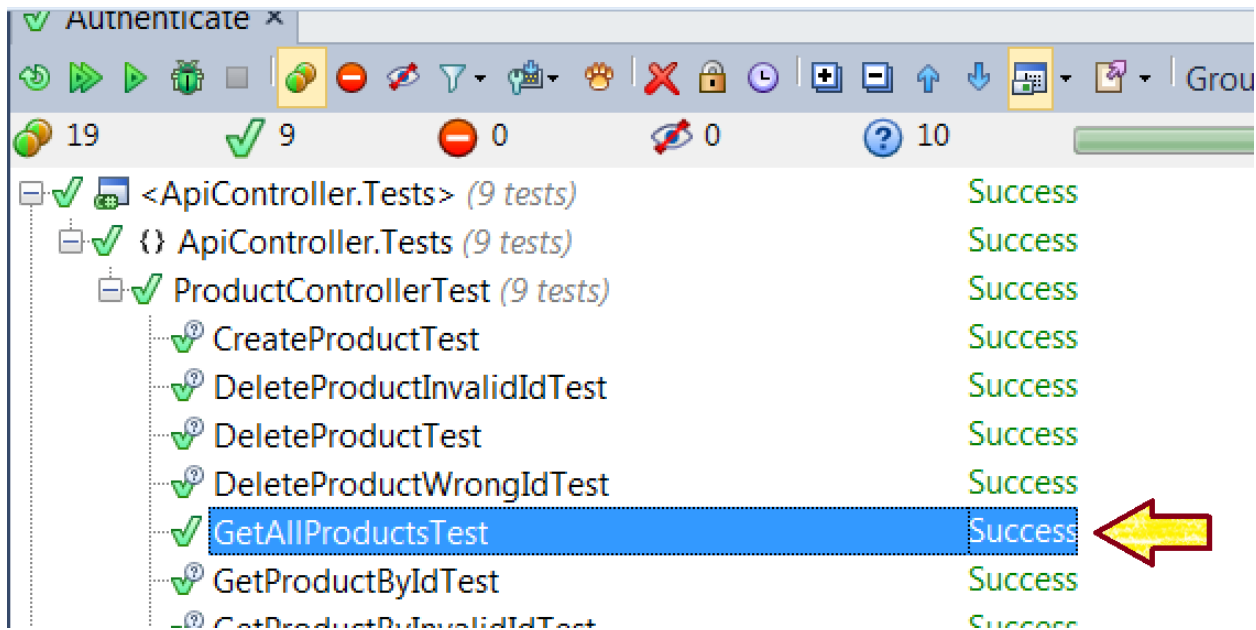
```
[Test]
public void GetAllProductsTest()
{
    var productController = new ProductController(_productService)
    {
        Request = new HttpRequestMessage
        {
            Method = HttpMethod.Get,
            RequestUri = new Uri(ServiceBaseUrl + "v1/Products/Product/a")
        }
    };
    productController.Request.Properties.Add(HttpPropertyKeys.HttpConfig);

    _response = productController.Get();

    var responseResult = JsonConvert.DeserializeObject<List<Product>>(_response.Content.ReadAsStringAsync().Result);
    Assert.AreEqual(responseResult.Count, 5, HttpStatusCode.OK);
    Assert.AreEqual(responseResult.Count, 5, HttpStatusCode.OK);
    var comparer = new ProductComparer();
    CollectionAssert.AreEqual(responseResult, _products, comparer);
}

// Response Data
// [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.

9. GetProductByIdTest ()

This unit test verifies if the correct result is returned if we try to invoke GetProductById() method of product controller.

[Test]

```
public void GetProductByIdTest()
{
    var productController = new ProductController(_productService)
    {
        Request = new HttpRequestMessage
        {
            Method = HttpMethod.Get,
            RequestUri = new Uri(ServiceBaseUrl + "v1/Products/Product/productid/2")
        }
    }
```



```
};

productController.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey, new
HttpConfiguration());

_response = productController.Get(2);

var responseResult =
JsonConvert.DeserializeObject<Product>(_response.Content.ReadAsStringAsync().Result);

Assert.AreEqual(_response.StatusCode, HttpStatusCode.OK);

AssertObjects.PropertyValuesAreEquals(responseResult,
_products.Find(a => a.ProductName.Contains("Mobile")));
}
```

I have used a sample product id “2” to test the method. Again we get the result in JSON format inside HttpResponseMessage that we de-serialize. First assert compares for the status code and second assert makes use of AssertObject class to compare the properties of the returned product with the actual “mobile” named product having product id as 2 from the list of products.


```

[Test]
public void GetProductByIdTest()
{
    var productController = new ProductController(_productService)
    {
        Request = new HttpRequestMessage
        {
            Method = HttpMethod.Get,
            RequestUri = new Uri(ServiceBaseUrl + "v1/Products/Product/productid/2")
        }
    };
    productController.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey, new HttpConfigura

    _response = productController.Get(2);

    var responseResult = JsonConvert.DeserializeObject<Product>(_response.Content.ReadAsStringAsync()
    Assert.AreEqual(responseResult.DataModel.Product.IsCode, OK);
    AssertObjects.ProductId 2 responseResult,
    AssertObjects.ProductName "Mobile" _products.Find(a => a.ProductName.Contains("Mobile")));
}

```

✓ DeleteProductInvalidIdTest	Success
✓ DeleteProductTest	Success
✓ DeleteProductWrongIdTest	Success
✓ GetAllProductsTest	Success
✓ GetProductByIdTest	Success
✓ GetProductByInvalidIdTest	Success
✓ GetProductByWrongIdTest	Success
✓ UpdateProductTest	Success

Testing Exceptions from WebAPI

NUnit provides flexibility to even test the exceptions. Now if we want to unit test the alternate exit point for `GetProductById()` method i.e. an exception so what should we do? Remember it was easy to test the alternate exit point for business services method because it returned null. Now in case of exception, NUnit provides an attribute `ExpectedException`. We can define the type of exception expected to be returned from the method call. Like if I make a call to the same method with wrong id, the expectation is that it should return an exception with `ErrorCode 1001` and an error description telling "No product found for this id."

So in our case the expected exception type is `ApiDataException` (got it from controller method). Therefore we can define the Exception attribute as `[ExpectedException("WebApi.ErrorHelper.ApiDataException")]`

And call the controller method with wrong id. But there is an alternate way to assert the exception. NUnit also provides us flexibility to assert the exception by `Assert.Throws`. This statement asserts the exception and returns that particular exception to the caller. Once we get that particular exception we can assert it with its `ErrorCode` and `ErrorDescription` or on whatever property you want to.

10. GetProductByWrongIdTest ()

```
[Test]

//[ExpectedException("WebApi.ErrorHelper.ApiDataException")]

public void GetProductByWrongIdTest()
{
    var productController = new ProductController(_productService)
    {
        Request = new HttpRequestMessage
        {
            Method = HttpMethod.Get,
            RequestUri = new Uri(ServiceBaseUrl + "v1/Products/Product/productid/10")
        }
    };

    productController.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey,
new HttpConfiguration());
```



```

var ex = Assert.Throws<ApiDataException>(() => productController.Get(10));

Assert.That(ex.ErrorCode, Is.EqualTo(1001));

Assert.That(ex.ErrorDescription, Is.EqualTo("No product found for this id.));

}

```

In the above code, I have commented out the Exception attribute approach and followed the alternate one.

I called the method with wrong id (that does not exists in our product list) in the statement,

```
var ex = Assert.Throws<ApiDataException>(() => productController.Get(10));
```

The above statement expects ApiDataException and stores the returned exception in “ex”.

Now we can assert the “ex” exception properties like ErrorCode and ErrorDescription with the actual desired result.



<ApiController.Tests> (9 tests)	Success
{ } ApiController.Tests (9 tests)	Success
ProductControllerTest (9 tests)	Success
CreateProductTest	Success
DeleteProductInvalidIdTest	Success
DeleteProductTest	Success
DeleteProductWrongIdTest	Success
GetAllProductsTest	Success
GetProductByIdTest	Success
GetProductByInvalidIdTest	Success
GetProductByWrongIdTest	Success
UpdateProductTest	Success



11. GetProductByInvalidIdTest ()

Another exit point for the same method is that if request for a product comes with an invalid id then an exception is thrown. Let's test that method for this scenario,

[Test]

```
// [ExpectedException("WebApi.ErrorHelper.ApiException")]
```

```
public void GetProductByInvalidIdTest()
```

```
{
```

```
  var productController = new ProductController(_productService)
```

```
  {
```

```
    Request = new HttpRequestMessage
```

```
    {
```

```
      Method = HttpMethod.Get,
```

```
      RequestUri = new Uri(ServiceBaseUrl + "v1/Products/Product/productid/-1")
```



```

    }

    };

    productController.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey,
new HttpConfiguration());

    var ex = Assert.Throws<ApiException>(() => productController.Get(-1));

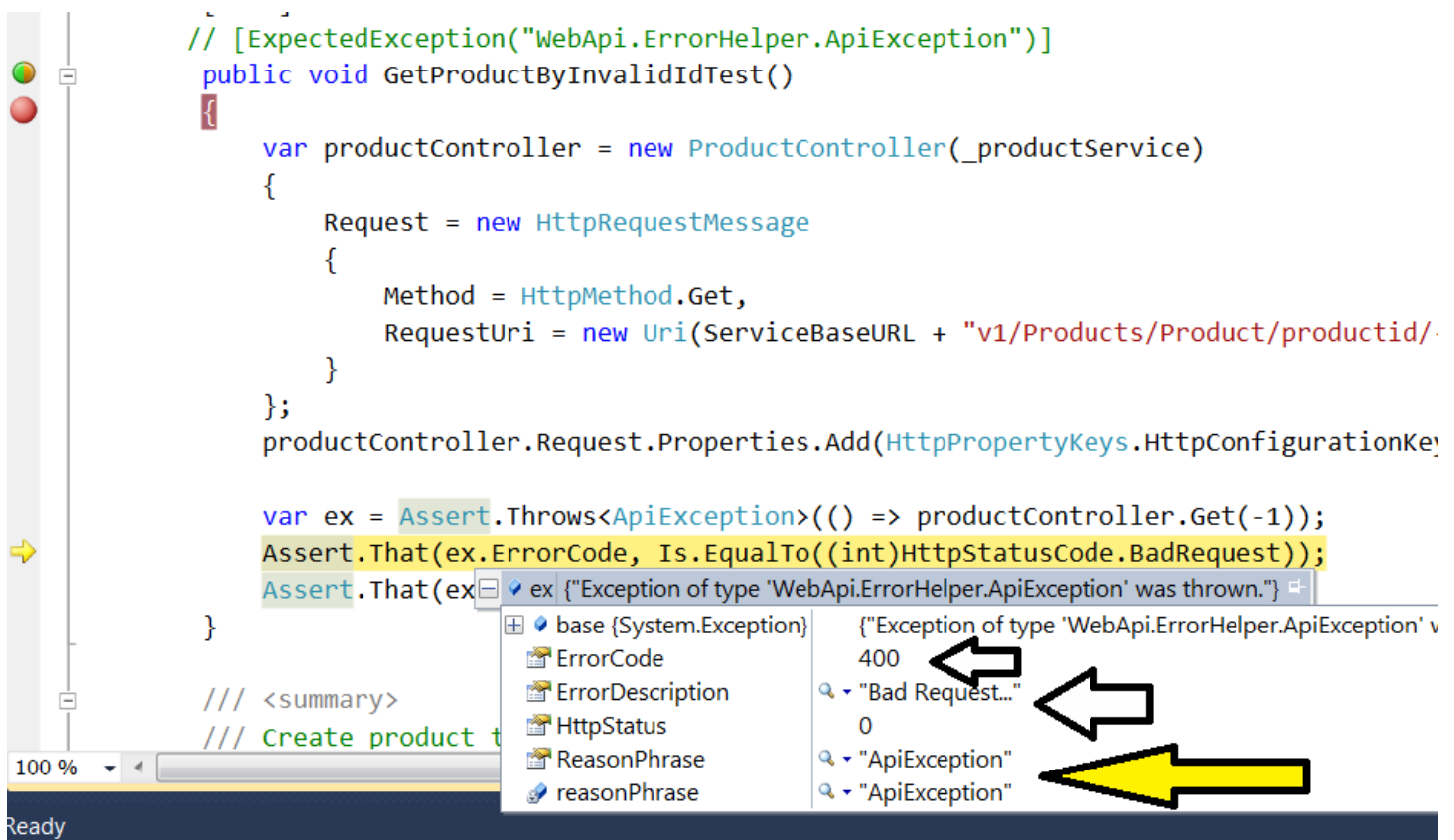
    Assert.That(ex.ErrorCode, Is.EqualTo((int)HttpStatusCode.BadRequest));

    Assert.That(ex.ErrorDescription, Is.EqualTo("Bad Request..."));

}


```

I passed an invalid id i.e. -1 to the controller method and it throws an exception of type ApiException with ErrorCode equal to HttpStatusCode.BadRequest and ErrorDescription equal to "bad Request...".



Test result,

✔ GetAllProductsTest	Success
✔ GetProductByIdTest	Success
✔ GetProductByInvalidIdTest	Success
✔ GetProductByWrongIdTest	Success



i.e. Passed. Other tests are very much of same kind like I explained.

12. CreateProductTest ()

```

/// <summary>
/// Create product test
/// </summary>

[Test]
public void CreateProductTest()
{
    var productController = new ProductController(_productService)
    {
        Request = new HttpRequestMessage
        {
            Method = HttpMethod.Post,
            RequestUri = new Uri(ServiceBaseURL + "v1/Products/Product/Create")
        }
    }

```



```

};

productController.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey,
new HttpConfiguration());

var newProduct = new ProductEntity()
{
    ProductName = "Android Phone"
};

var maxProductIDBeforeAdd = _products.Max(a => a.ProductId);

newProduct.ProductId = maxProductIDBeforeAdd + 1;

productController.Post(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));
}

```

13. UpdateProductTest ()

```

/// <summary>

/// Update product test

/// </summary>

[Test]

```



```

public void UpdateProductTest()
{
    var productController = new ProductController(_productService)
    {
        Request = new HttpRequestMessage
        {
            Method = HttpMethod.Put,
            RequestUri = new Uri(ServiceBaseURL + "v1/Products/Product/Modify")
        }
    };

    productController.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey,
new HttpConfiguration());

    var firstProduct = _products.First();

    firstProduct.ProductName = "Laptop updated";

    var updatedProduct = new ProductEntity() { ProductName =
firstProduct.ProductName, ProductId = firstProduct.ProductId };

    productController.Put(firstProduct.ProductId, updatedProduct);

    Assert.That(firstProduct.ProductId, Is.EqualTo(1)); // hasn't changed
}

```

14. DeleteProductTest ()


```
/// <summary>

/// Delete product test

/// </summary>

[Test]

public void DeleteProductTest()

{

    var productController = new ProductController(_productService)

    {

        Request = new HttpRequestMessage

        {

            Method = HttpMethod.Put,

            RequestUri = new Uri(ServiceBaseUrl + "v1/Products/Product/Remove")

        }

    };

    productController.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey,

new HttpConfiguration());

    int maxID = _products.Max(a => a.ProductId); // Before removal

    var lastProduct = _products.Last();

    // Remove last Product

    productController.Delete(lastProduct.ProductId);
```



```
Assert.That(maxID, Is.GreaterThan(_products.Max(a => a.ProductId))); // Max id
reduced by 1
```

```
}
```

15. DeleteInvalidProductTest ()

```
/// <summary>
```

```
/// Delete product test with invalid id
```

```
/// </summary>
```

```
[Test]
```

```
public void DeleteProductInvalidIdTest()
```

```
{
```

```
var productController = new ProductController(_productService)
```

```
{
```

```
Request = new HttpRequestMessage
```

```
{
```

```
Method = HttpMethod.Put,
```

```
RequestUri = new Uri(ServiceBaseUrl + "v1/Products/Product/remove")
```

```
}
```

```
};
```

```
productController.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey,
new HttpConfiguration());
```

```
var ex = Assert.Throws<ApiException>(() => productController.Delete(-1));
```



```

    Assert.That(ex.ErrorCode, Is.EqualTo((int)HttpStatusCode.BadRequest));

    Assert.That(ex.ErrorDescription, Is.EqualTo("Bad Request..."));
}

```

16. DeleteProductWithWrongIdTest ()

```

/// <summary>
/// Delete product test with wrong id
/// </summary>

[Test]
public void DeleteProductWrongIdTest()
{
    var productController = new ProductController(_productService)
    {
        Request = new HttpRequestMessage
        {
            Method = HttpMethod.Put,
            RequestUri = new Uri(ServiceBaseURL + "v1/Products/Product/remove")
        }
    };

    productController.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey,
    new HttpConfiguration());

    int maxID = _products.Max(a => a.ProductId); // Before removal

```



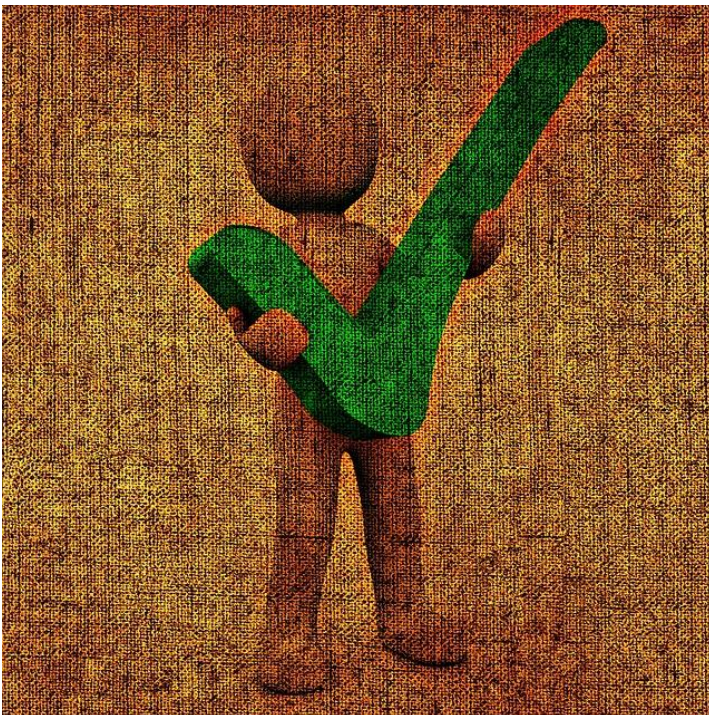
```
var ex = Assert.Throws<ApiDataException>(() =>
productController.Delete(maxID+1));

Assert.That(ex.ErrorCode, Is.EqualTo(1002));

Assert.That(ex.ErrorDescription, Is.EqualTo("Product is already deleted or not exist in
system."));
}
```

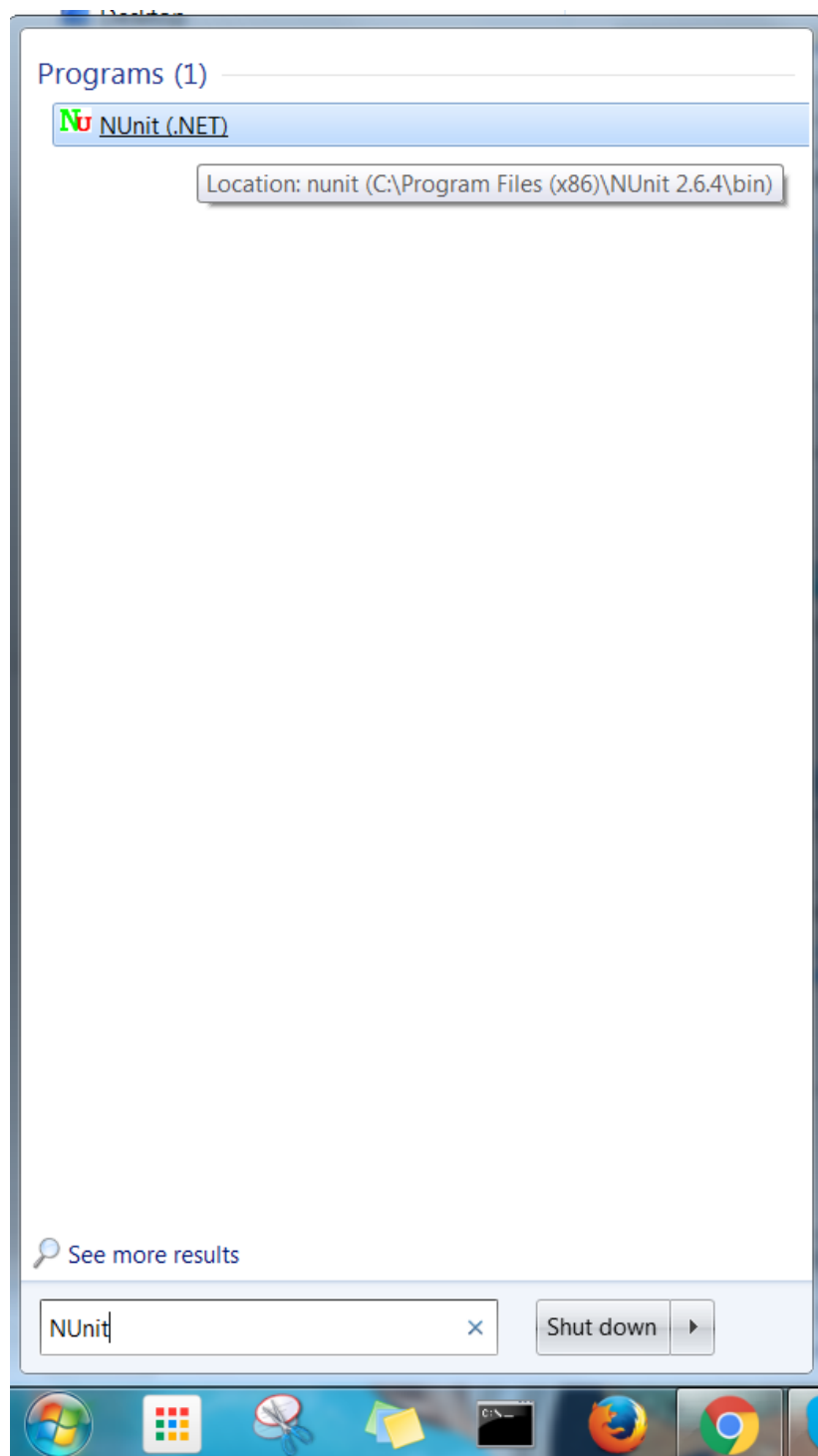
All the above mentioned tests are self-explanatory and are more like how we tested BusinessServices. the idea was to explain how we write tests in WebAPI. Let's run all the tests through NUnit UI.

Test through NUnit UI



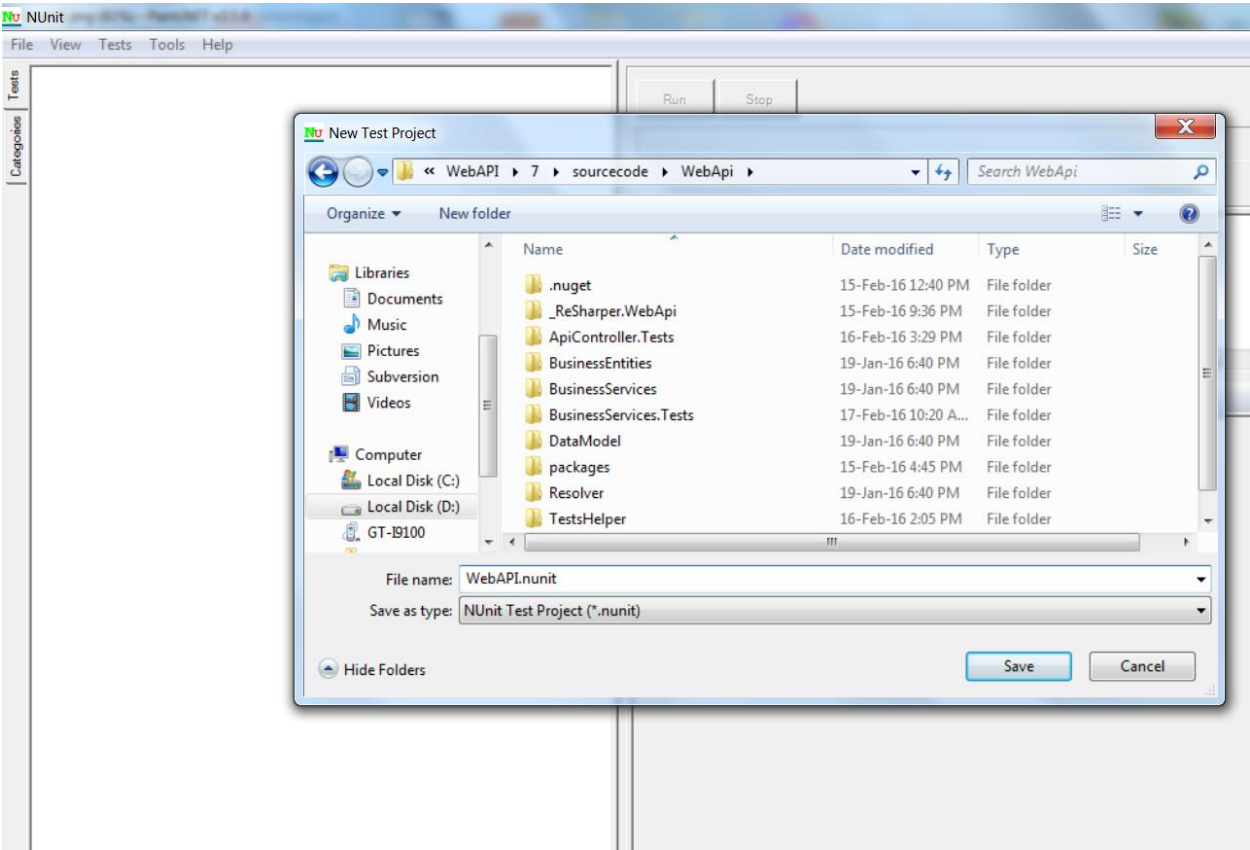
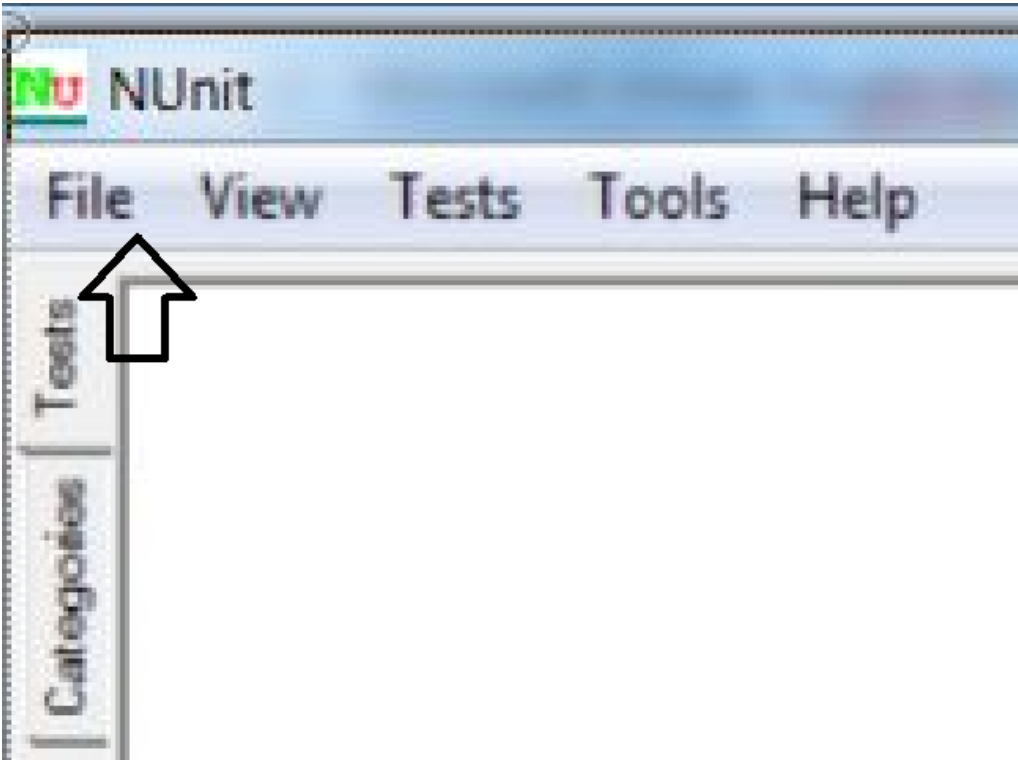
6. Step 1:

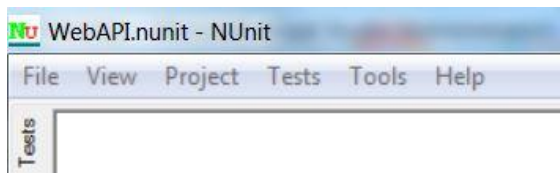
Launch NUnit UI. I have already explained how to install NUnit on the windows machine. Just launch the NUnit interface with its launch icon,



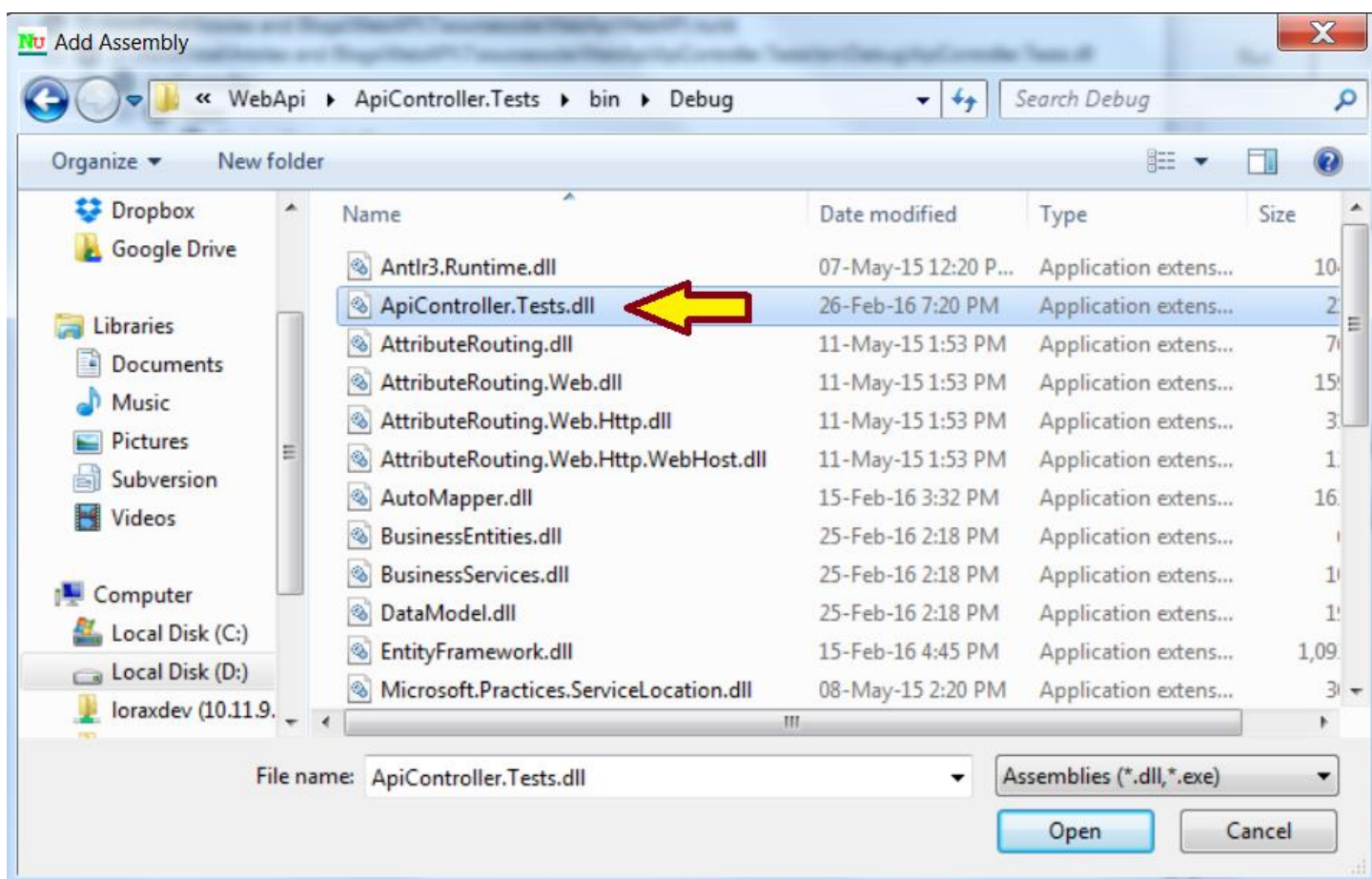
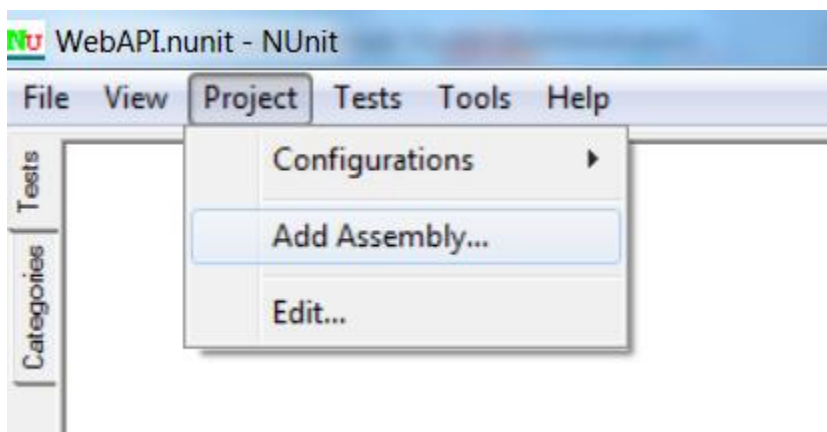
7. Step 2 :

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

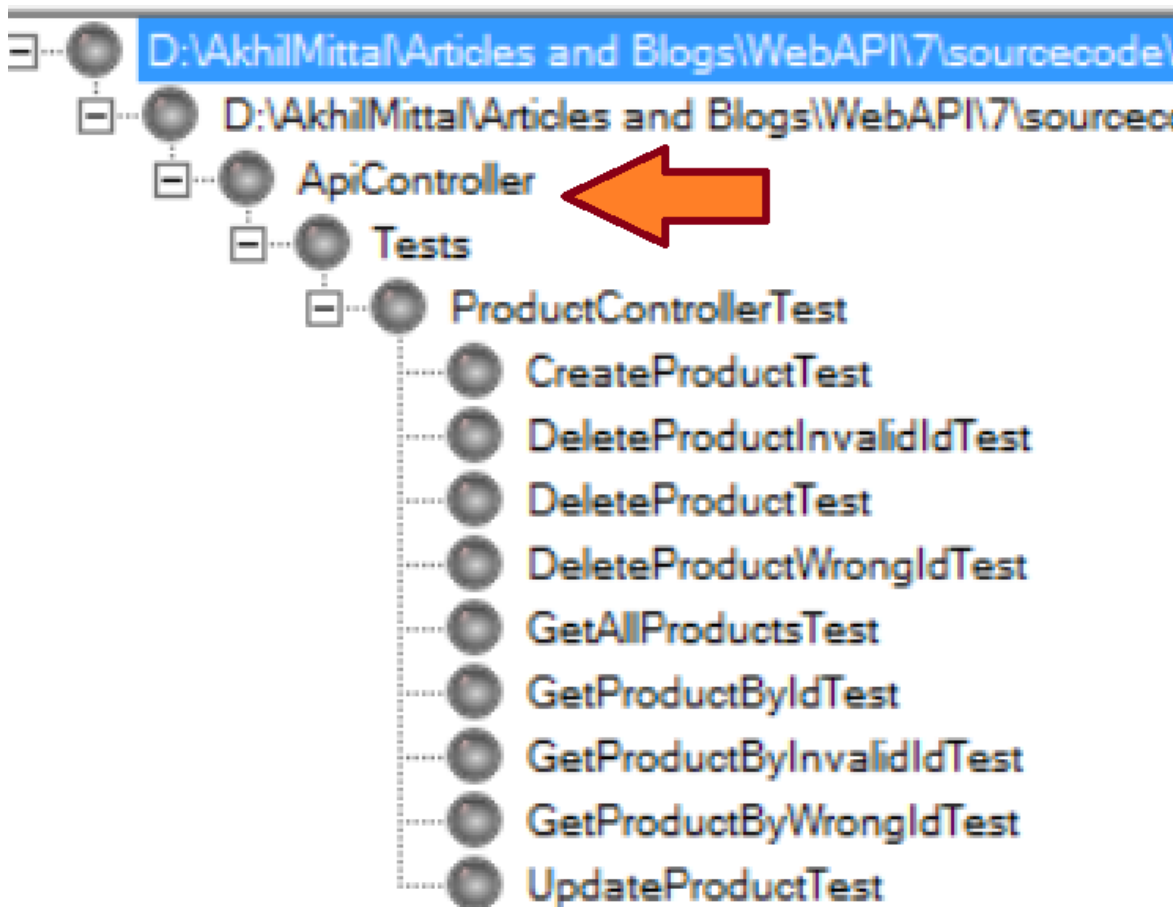




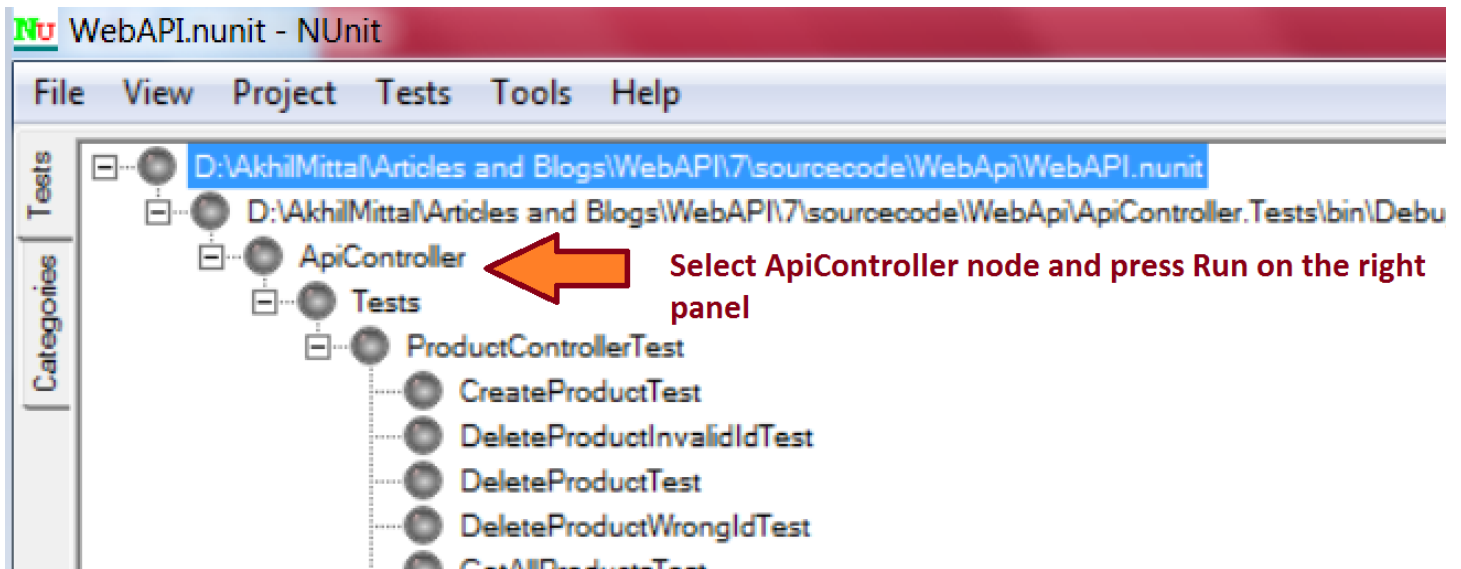
8. Step 3: Now, click on Project-> Add Assembly and browse for ApiController.Tests.dll (The library created for your unit test project when compiled)



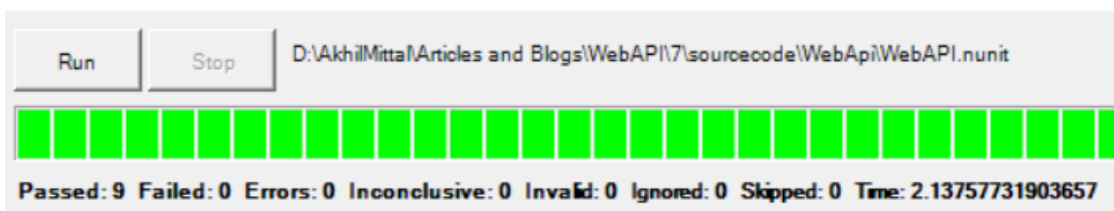
9. Step 4: Once the assembly is browsed, you'll see all the unit tests for that test project gets loaded in the UI and are visible on the interface.



10. At the right hand side panel of the interface, you'll see a Run button that runs all the tests of Api controller. Just select the node ApiController 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.



Integration Tests

I'll give just an idea of what integration tests are and how can we write it. Integration tests doesn't run in memory. For WebAPI's the best practice to write integration test is when the WebAPI is self hosted. You can try writing integration test when you host an API, so that you get an actual URL or endpoint of the service you want to test. The test is performed on actual data and actual service. Let's proceed with an example. I have hosted my web api and I want to test GetAllProducts() method of WebAPI. My hosted URL for the particular controller action is <http://localhost:50875/v1/Products/Product/allproducts>.

Now I know that I am not going to test my controller method through dll reference but I want to actually test its endpoint for which I need to pass an authentication token because that end point is secured and can not be authorized until I add a secure token to the Request header. Following is the integration test for GetAllProducts().

[Test]

```
public void GetAllProductsIntegrationTest()
```

```
{
```

#region To be written inside Setup method specifically for integration tests

```
var client = new HttpClient { BaseAddress = new Uri(ServiceBaseURL) };
```

```
client.DefaultRequestHeaders.Add("Authorization", "Basic YWtoaWw6YWtoaWw=");
```

```
MediaTypeFormatter jsonFormatter = new JsonMediaTypeFormatter();
```

```
_response = client.PostAsync("login", null).Result;
```

```
if (_response != null && _response.Headers != null &&
_response.Headers.Contains("Token") && _response.Headers.GetValues("Token") != null)
```

```
{
```

```
client.DefaultRequestHeaders.Clear();
```

```
_token = ((string[])(_response.Headers.GetValues("Token")))[0];
```



```

client.DefaultRequestHeaders.Add("Token", _token);

}

#endregion

_response = client.GetAsync("v1/Products/Product/allproducts/").Result;

var responseResult =

JsonConvert.DeserializeObject<List<ProductEntity>>(_response.Content.ReadAsStringAsync
()).Result);

Assert.AreEqual(_response.StatusCode, HttpStatusCode.OK);

Assert.AreEqual(responseResult.Any(), true);

}

```

I have used same class to write this test, but you should always keep your unit tests segregated from integration tests, so use another test project to write integration tests for web api. First we have to request a token and add it to client request,

```

var client = new HttpClient { BaseAddress = new Uri(ServiceBaseURL) };

client.DefaultRequestHeaders.Add("Authorization", "Basic YWtoaWw6YWtoaWw=");

MediaTypeFormatter jsonFormatter = new JsonMediaTypeFormatter();

_response = client.PostAsync("login", null).Result;

if (_response != null && _response.Headers != null &&
_response.Headers.Contains("Token") && _response.Headers.GetValues("Token") != null)

{

client.DefaultRequestHeaders.Clear();

```



```
_token = ((string[])(_response.Headers.GetValues("Token")))[0];  
client.DefaultRequestHeaders.Add("Token", _token);  
}
```

In above code I have initialized the client with running service's base URL i.e. <http://localhost:50875>. After initialization I am setting a default request header to call my login endpoint of Authentication controller to fetch valid token. Once the user logs in with his credentials he get a valid token. To read in detail about security refer my article on [security in web api](#). I have passed base 64 string of my credentials username :Akhil and password:Akhil for basic authentication. Once request gets authenticated, I get a valid token in `_response.Headers` that I fetch and assign to `_token` variable and add to client's default header with this line of code,

```
_token = ((string[])(_response.Headers.GetValues("Token")))[0];  
client.DefaultRequestHeaders.Add("Token", _token);
```

Then I am calling the actual service URL from the same client,

```
_response = client.GetAsync("v1/Products/Product/allproducts/").Result;
```

And we get the result as success. Follow the screen shots.

Step1: Get Token


```

public void GetAllProductsIntegrationTest()
{
    #region To be written inside Setup method specifically for integration te
    var client = new HttpClient { BaseAddress = new Uri(ServiceBaseURL) };
    client.DefaultRequestHeaders.Add("Authorization", "Basic YWtoYWw6YWtoYWw=
    MediaTypeFormatter jsonFormatter = new JsonMediaTypeFormatter();
    _response = client.PostAsync("login", null).Result;

    if (_response != null && _response.Headers != null && _response.Headers.C
    {
        client.DefaultRequestHeaders.Clear();
        _token = ((string[])(_response.Headers.GetValues("Token")))[0];
        cli _token "4bffc06f-d8b1-4eda-b4e6-df9568dd53b1"
    }
}

```

We got the token : **4bffc06f-d8b1-4eda-b4e6-df9568dd53b1**. Now since this is a real time test. This token should get saved in database. Let's check.

Step2 : Check database

	TokenId	UserId	AuthToken	IssuedOn	ExpiresOn
▶	17	1	4bffc06f-d8b1-4eda-b4e6-df9568dd53b1	2016-02-26 23:...	2016-02-27 00:...
*	NULL	NULL	NULL	NULL	NULL

We got the same token in database. It proves we are testing on real live URL.

Step3 : Check ResponseResult

```

if (_response != null && _response.Headers != null && _respo
{
    client.DefaultRequestHeaders.Clear();
    _token = ((string[])(_response.Headers.GetValues("Token"
    client.DefaultRequestHeaders.Add("Token", _token);
}
#endregion

_response = client.GetAsync("v1/Products/Product/allproducts.
var responseResult =
    JsonConvert.DeserializeObject<ProductEntity>>(_resp
Assert.AreEqual([0], {BusinessEntities.ProductEntity} de.OK);
Assert.AreEqual(
}
#endregion

```


Here we got the response result with 6 products where first product id is 1 and product name is “Laptop”. Check the database for complete product list,

Results

Messages

	ProductId	ProductName
1	1	Laptop
2	2	computer
3	4	IPhone
4	5	Bag
5	6	Watch
6	8	sample string 2

We get the same data. This proves our test is a success.

✓ <ApiController.Tests> (10 tests)	Success
✓ {} ApiController.Tests (10 tests)	Success
✓ ProductControllerTest (10 tests)	Success
✓ CreateProductTest	Success
✓ DeleteProductInvalidIdTest	Success
✓ DeleteProductTest	Success
✓ DeleteProductWrongIdTest	Success
✓ GetAllProductsIntegrationTest	Success
✓ GetAllProductsTest	Success
✓ GetProductByIdTest	Success
✓ GetProductByInvalidIdTest	Success
✓ GetProductByWrongIdTest	Success

Likewise you can write more integration tests.

Difference between Unit tests and Integration tests

I'll not write much, but wanted to share one of my good readings on this from this [reference link](#)

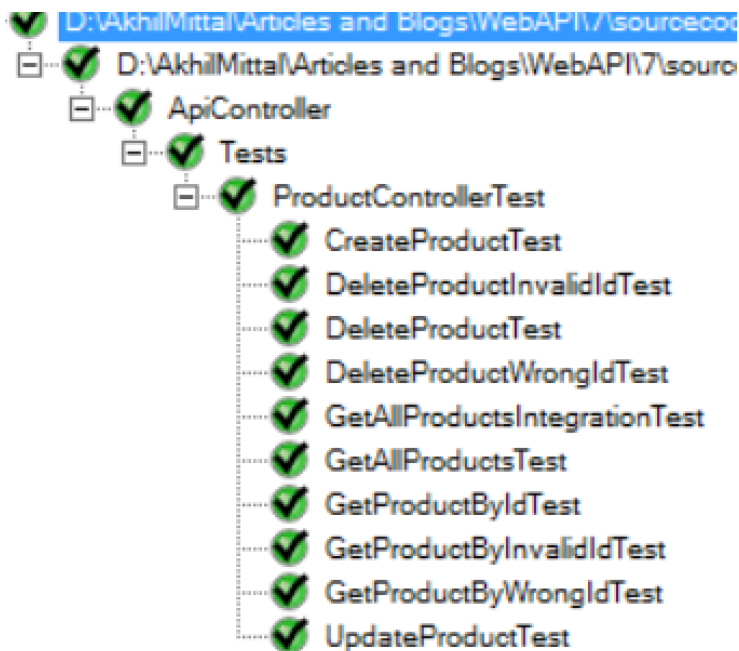
Unit Testing	Integration Testing
--------------	---------------------

Unit testing is a type of testing to check if the small piece of code is doing what it is supposed to do.	Integration testing is a type of testing to check if different pieces of the modules are working together.
Unit testing checks a single component of an application.	The behavior of integration modules is considered in the Integration testing.
The scope of Unit testing is narrow, it covers the Unit or small piece of code under test. Therefore while writing a unit test shorter codes are used that target just a single class.	The scope of Integration testing is wide, it covers the whole application under test and it requires much more effort to put together.
Unit tests should have no dependencies on code outside the unit tested.	Integration testing is dependent on other outside systems like databases, hardware allocated for them etc.
This is first type of testing is to be carried out in Software testing life cycle and generally executed by developer.	This type of testing is carried out after Unit testing and before System testing and executed by the testing team.
Unit testing is not further sub divided into different types.	Integration testing is further divided into different types as follows:
	Top-down Integration, Bottom-Up Integration and so on.
Unit testing is starts with the module specification.	Integration testing is starts with the interface specification.

The detailed visibility of the code is comes under Unit testing.	The visibility of the integration structure is comes under Integration testing.
Unit testing mainly focus on the testing the functionality of individual units only and does not uncover the issues arises when different modules are interacting with each other.	Integration testing is to be carried out to discover the the issues arise when different modules are interacting with each other to build overall system.
The goal of Unit testing is to test the each unit separately and ensure that each unit is working as expected.	The goal of Integration testing is to test the combined modules together and ensure that every combined module is working as expected.
Unit testing comes under White box testing type.	Integration testing is comes under both Black box and White box type of testing.

Reference : <http://www.softwaretestingclass.com/what-is-difference-between-unit-testing-and-integration-testing/>

Conclusion



In this chapter we learnt how to write unit tests for Web API controller and primarily on basic CRUD 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. We also learned about how to write integration tests for WebAPI endpoints. I hope this was useful to you. You can download the complete source code of this article with packages from [GitHub](#). Happy coding 😊

OData in ASP.NET Web APIs

OData

OData is a protocol that provides a flexibility of creating query able REST services. It provides certain query options through which the on demand data can be fetched from the server by the client over HTTP.

Following is the definition from [asp.net](#),

“The Open Data Protocol (OData) is a data access protocol for the web. OData provides a uniform way to query and manipulate data sets through CRUD operations (create, read, update, and delete).”

More elaborated [from](#),

“OData defines parameters that can be used to modify an OData query. The client sends these parameters in the query string of the request URI. For example, to sort the results, a client uses the \$orderby parameter:

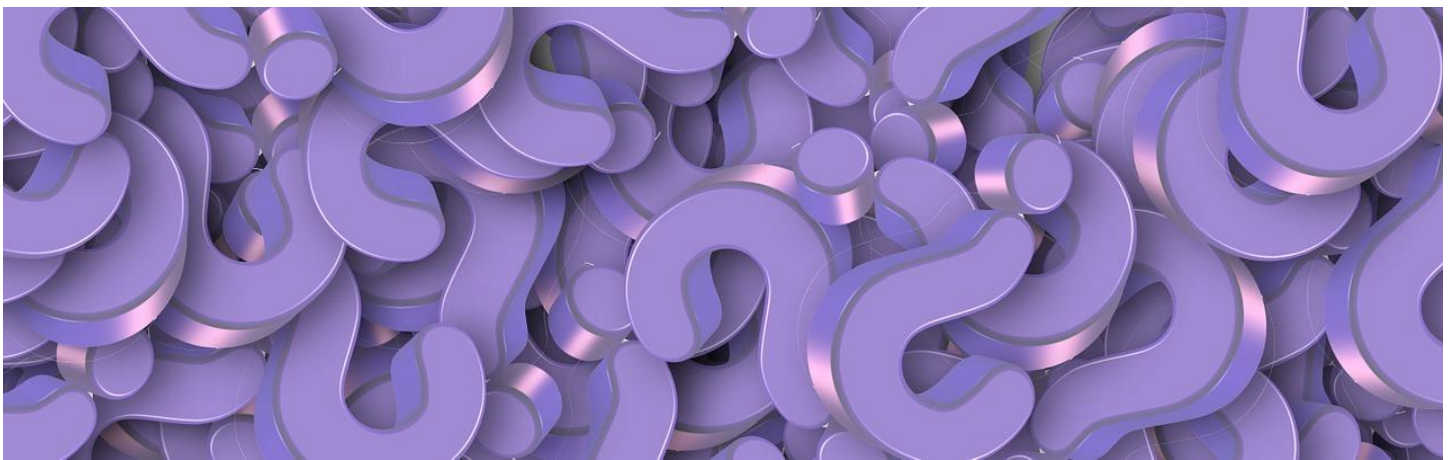
`http://localhost/Products?$orderby=Name`

The OData specification calls these parameters query options. You can enable OData query options for any Web API controller in your project — the controller does not need to be an OData endpoint. This gives you a convenient way to add features such as filtering and sorting to any Web API application.

”

Suppose our product table in the database contains more than 50000 products and we want to fetch only top 50 products based on certain conditions like product id or price or name, then as per our current implementation of the service, I'll have to fetch all the products from the server database and filter them on client or another option could be that I fetch the data at server only and filter the same and send the filtered data to client. In both the cases I am bearing a cost of writing an extra code of filtering the data. Here comes

OData in picture. OData allows you to create services that are query able. If the endpoints of the exposed services are OData enabled or supports OData query options then the service implementation would be in such a way that it considers the OData request and process it accordingly. So had that request for 50 records been an OData request, the service would have fetched only 50 records from the server. Not only filtering, but OData provides features like searching, sorting, skipping the data, selecting the data too. I'll explain the concept with practical implementation. We'll use our already created service and modify them to be enabled for OData query options.



Query Options

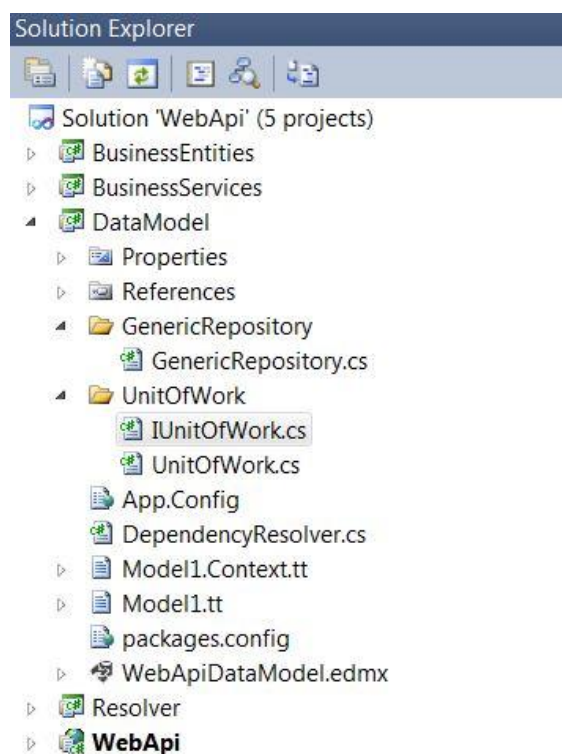
Following are the OData query options that asp.net WebAPI supports,

1. **\$orderby:** Sorts the fetched record in particular order like ascending or descending.
2. **\$select:** Selects the columns or properties in the result set. Specifies which all attributes or properties to include in the fetched result.
3. **\$skip:** Used to skip the number of records or results. For e.g. I want to skip first 100 records from the database while fetching complete table data, then I can make use of \$skip.
4. **\$top:** Fetches only top n records. For e.g. I want to fetch top 10 records from the database, then my particular service should be OData enabled to support \$top query option.
5. **\$expand:** Expands the related domain entities of the fetched entities.

6. **\$filter**: Filters the result set based on certain conditions, it is like where clause of LINQ. For e.g. I want to fetch the records of 50 students who have scored more than 90% marks, and then I can make use of this query option.
7. **\$inlinecount**: This query option is mostly used for pagination at client side. It tells the count of total entities fetched from the server to the client.

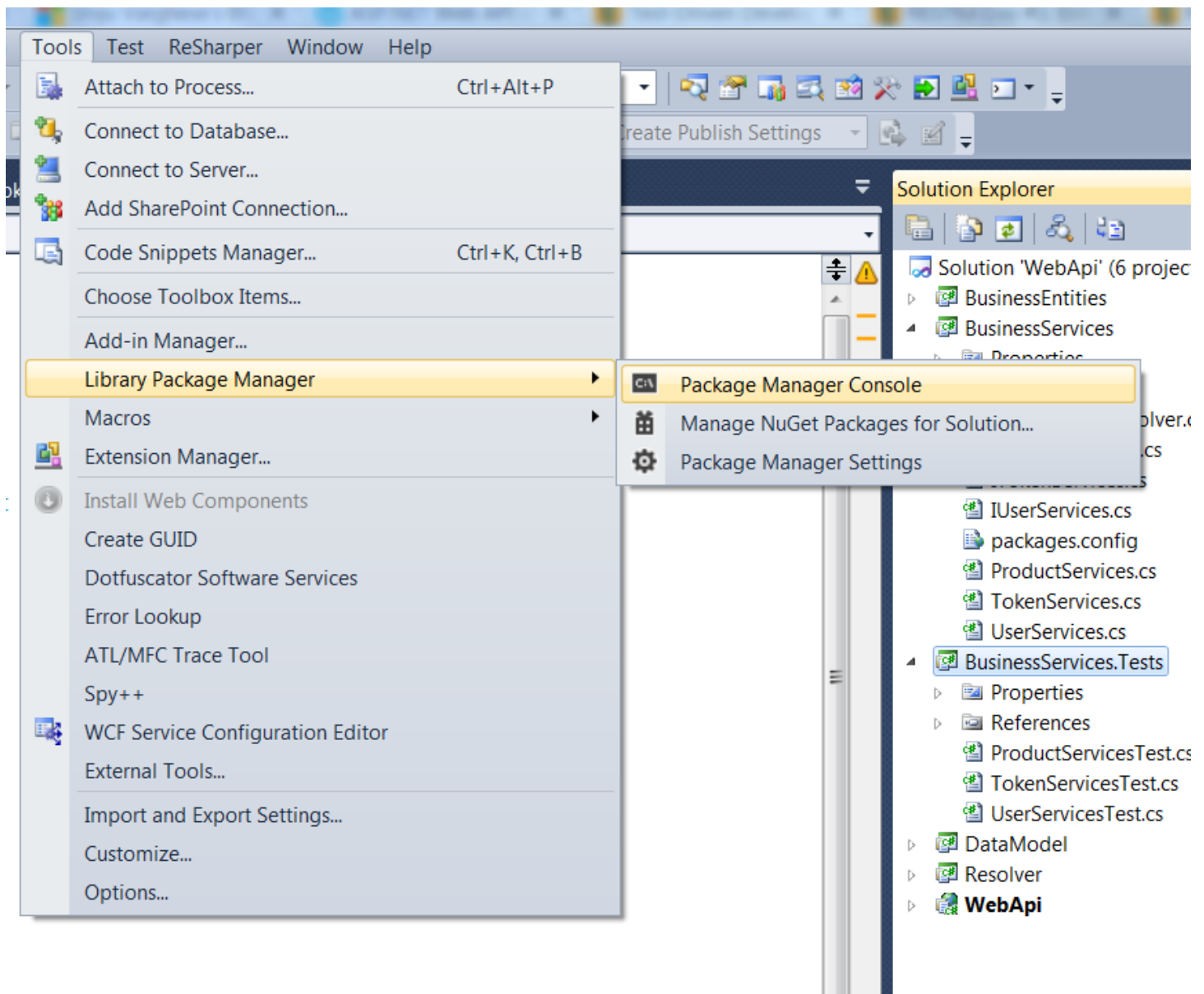
Setup Solution

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,



The solution contains the WebAPI application and related projects.

Step1: Click on Tools-> Library Package manager-> Package manager console



Step2: In Package manager console, select default project as WebApi and run the command: `Install-Package Microsoft.AspNet.WebApi.OData -Version 4.0.0`

Note that, since we are using VS 2010 and .Net framework 4.0, we need to install OData libraries compatible to it.

Package Manager Console

Package source: nuget.org

Default project: WebApi

```
PM> Install-Package Microsoft.AspNet.WebApi.OData -Version 4.0.0
Attempting to resolve dependency 'Microsoft.Net.Http (≥ 2.0.20710.0 && < 2.1)'.
Attempting to resolve dependency 'Microsoft.AspNet.WebApi.Client (≥ 4.0.20710.0 && < 4.1)'.
```

The command will download few dependent packages and reference the dll in your project references. You'll get the OData reference dll in your project references,

References

- Antlr3.Runtime
- AttributeRouting
- AttributeRouting.Web
- AttributeRouting.Web.Http
- AttributeRouting.Web.Http.WebHost
- BusinessEntities
- BusinessServices
- EntityFramework
- Microsoft.CSharp
- Microsoft.Data.Edm
- Microsoft.Data.OData
- Microsoft.Practices.ServiceLocation

Our project is set to make OData endpoints. You can create new services. I'll modify my existing services to demonstrate the OData working.

OData Endpoints

Open the ProductController class in WebAPI project and got to Get() method. This method fetches all the product records from the database. Following is the code,

```
[GET("allproducts")]

[GET("all")]

public HttpResponseMessage Get()
{
    var products = _productServices.GetAllProducts();

    var productEntities = products as List<ProductEntity> ?? products.ToList();

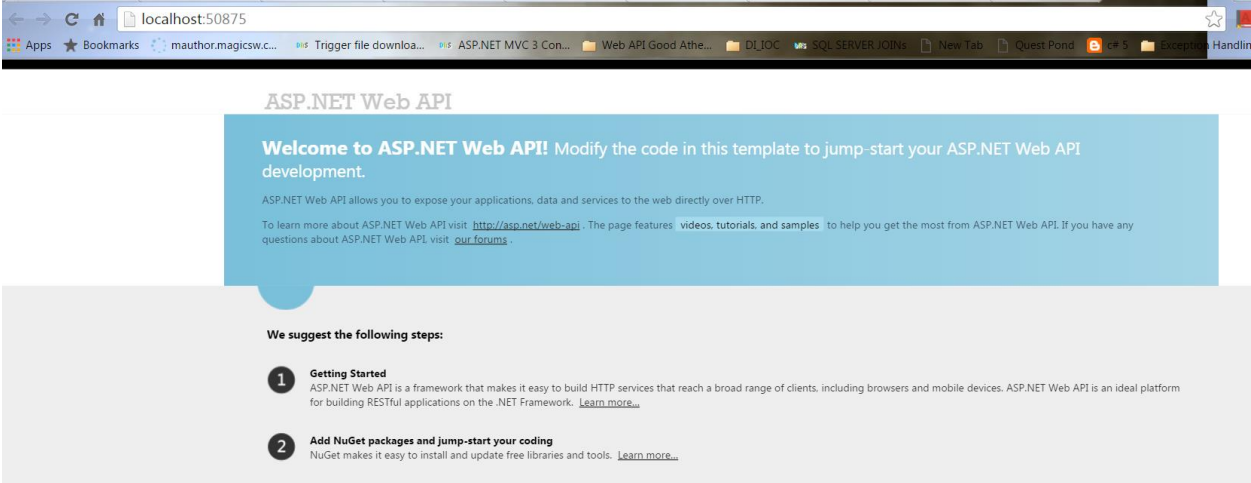
    if (productEntities.Any())

        return Request.CreateResponse(HttpStatusCode.OK, productEntities);

    throw new ApiDataException(1000, "Products not found",
HttpStatusCode.NotFound);
}
```

Let's run the code through test client,

Just run the application, we get,



Append/help in the URL and press enter, you’ll see the test client.

ASP.NET Web API Help Page

Introduction

Provide a general description of your APIs here.

Authenticate

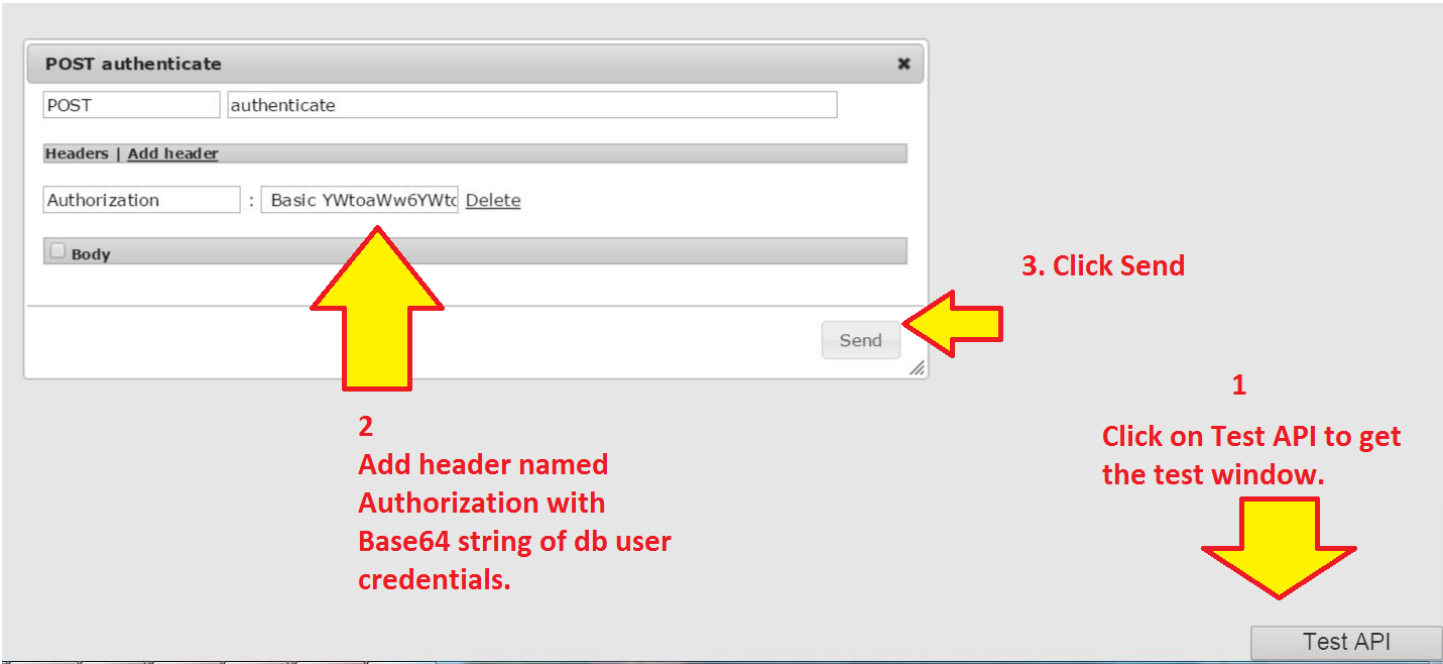
API	Description
POST authenticate	Documentation for 'Authenticate'.
POST login	Documentation for 'Authenticate'.
POST get/token	Documentation for 'Authenticate'.

Product

API	Description
GET v1/Products/Product/allproducts	Documentation for 'Get'.
GET v1/Products/Product/allproducts?id={id}	Documentation for 'Get'.
GET v1/Products/Product/all	Documentation for 'Get'.
GET v1/Products/Product/all?id={id}	Documentation for 'Get'.

Since our product Controller is secured, we need to get an authenticated token from the service and use the same to access product Controller methods. To read about WebAPI

security, refer [this article](#). Click on POST authenticate API method and get the TestAPI page of test client,



Let’s send the request with credentials now. Just add a header too with the request. Header should be like ,

Authorization : Basic YWtoaWw6YWtoaWw=

Here "YWtoaWw6YWtoaWw=" is my Base64 encoded user name and password in database i.e. akhil:akhil

If authorized, you’ll get a Token.Just save that token for making further calls to product Controller.

Now open your product controller’s “allproducts” endpoint in test client,

Product

API	Description
GET v1/Products/Product/allproducts	Documentation for 'Get'.

Test the endpoint,

GET v1/Products/Product/allproducts

GET

v1/Products/Product/allproducts

Headers | Add header

Token

:

f2b39f7e-8b69-45ee-b

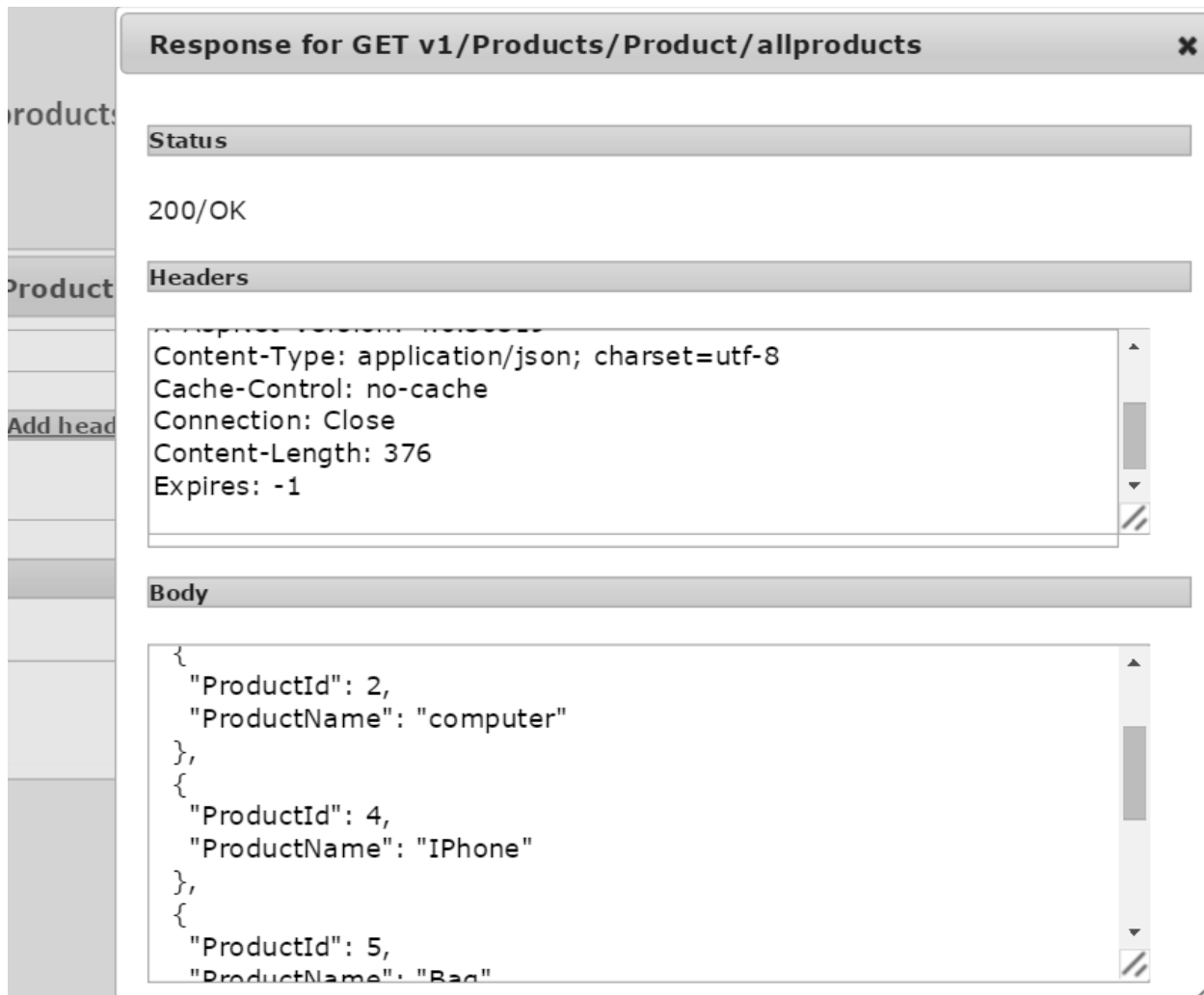
Delete

☐ Body

Send

We get response with all the 6 products,

©2016 Akhil Mittal (www.codet Teddy.com)



I'll use this controller method and make it OData endpoint and perform several query options over it.

Add an attribute named `[Queryable]` above the method and in `Request.CreateResponse` mark the `productEntities` to `productEntities.AsQueryable()`.

`[Queryable]`

`[GET("allproducts")]`

`[GET("all")]`

`public HttpResponseMessage Get()`


```
{  
  
    var products = _productServices.GetAllProducts().AsQueryable();  
  
    var productEntities = products as List<ProductEntity> ?? products.ToList();  
  
    if (productEntities.Any())  
  
        return Request.CreateResponse(HttpStatusCode.OK,  
productEntities.AsQueryable());  
  
    throw new ApiDataException(1000, "Products not found",  
HttpStatusCode.NotFound);  
  
}
```


\$top

Now test the API with **\$top** query option,

GET v1/Products/Product/allproducts ×

GET

v1/Products/Product/allproducts?\$top=2

 Append ?\$top=2 to service endpoint

Headers | Add header

Token

:

f2b39f7e-8b69-45ee-b

Delete

☐ **Body**

Send

Here in the above endpoint, I have just appended “**?\$top=2**” in the endpoint of the service i.e. like we append query strings. This statement means that I want to fetch only top two products from the service and the result is,

Response for GET v1/Products/Product/allproducts ✕

Status

200/OK

Headers

Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 127
Expires: -1

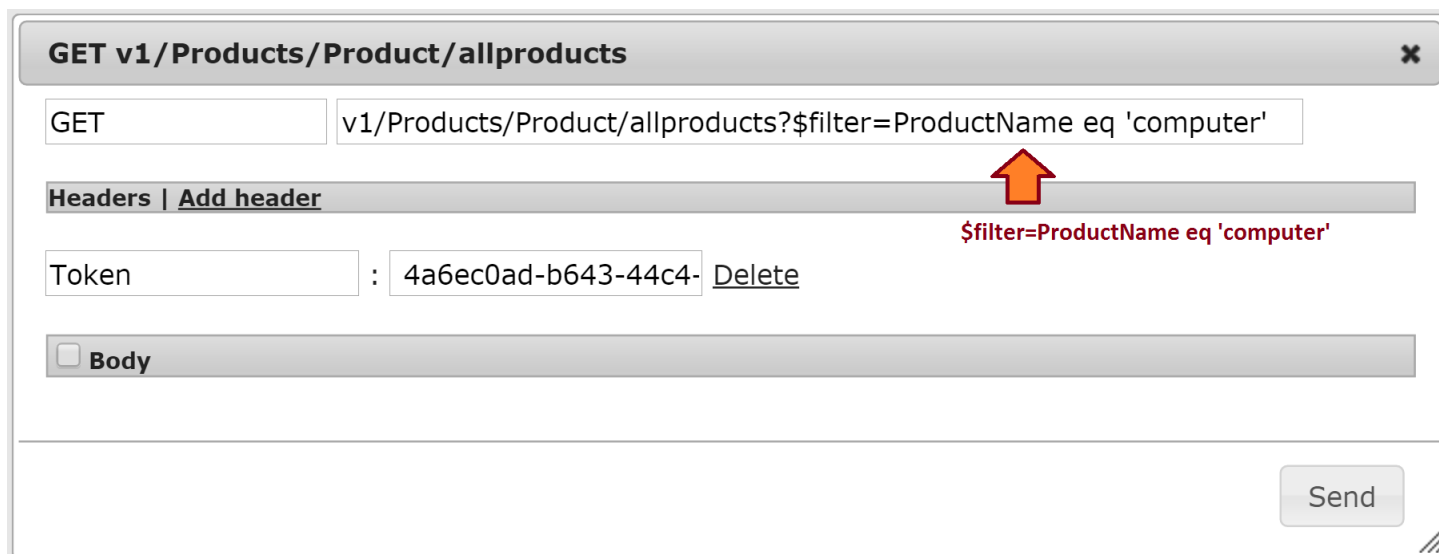
Body

```
[
  {
    "ProductId": 1,
    "ProductName": "Laptop"
  },
  {
    "ProductId": 2,
    "ProductName": "computer"
  }
]
```

We get only two products. So you can see here that it was very simple to make a service endpoint query able, and we did not have to write a new service to achieve this kind of result. Let us try few more options.

\$filter

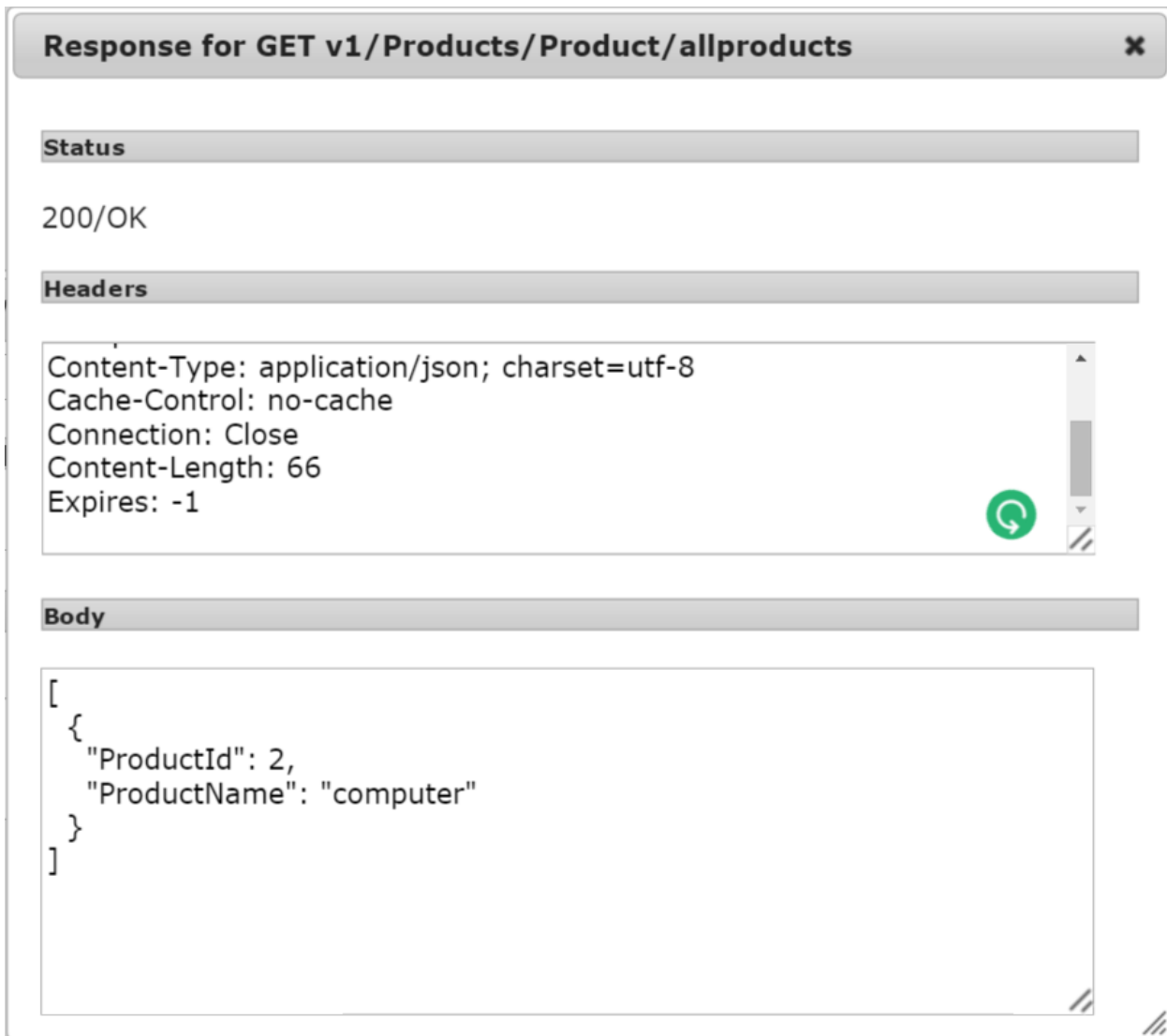
You can perform all the filtering over the records with this option. Let us try \$filter query option. Suppose we need to fetch all the products whose name is “computer”. You can use the same endpoint with filtering as shown below.



The screenshot shows an API client interface with the following components:

- Method:** GET
- URL:** v1/Products/Product/allproducts?\$filter=ProductName eq 'computer'
- Headers:** A section with a tab labeled "Headers | Add header". Below it, a header is defined: Token : 4a6ec0ad-b643-44c4- Delete. A red arrow points from the text "\$filter=ProductName eq 'computer'" to the URL.
- Body:** A section with a checkbox labeled "Body".
- Buttons:** A "Send" button is located at the bottom right.

I used **\$filter=ProductName eq 'computer'** as a query string, which means fetching product having product name “computer”, as a result we get only one record from products list because there was only one record having product name as “computer”.



You can use filter in many different ways as shown below,

Return all products with name equal to “computer”.

[http://localhost:50875/v1/Products/Product/allproducts?\\$filter=ProductName eq "computer"](http://localhost:50875/v1/Products/Product/allproducts?$filter=ProductName eq \)

Return all products with id less than 3.

[http://localhost:50875/v1/Products/Product/allproducts?\\$filter=ProductId lt 3](http://localhost:50875/v1/Products/Product/allproducts?$filter=ProductId lt 3)

Response for GET v1/Products/Product/allproducts ✕

Status

200/OK

Headers

Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 127
Expires: -1

Body

```
[
  {
    "ProductId": 1,
    "ProductName": "Laptop"
  },
  {
    "ProductId": 2,
    "ProductName": "computer"
  }
]
```

Logical operators: Return all products where id \geq 3 and id \leq 5.

[http://localhost:50875/v1/Products/Product/allproducts?\\$filter=ProductId ge 3 and ProductId le 5](http://localhost:50875/v1/Products/Product/allproducts?$filter=ProductId ge 3 and ProductId le 5)

Body

```
[
  {
    "ProductId": 4,
    "ProductName": "iPhone"
  },
  {
    "ProductId": 5,
    "ProductName": "Bag"
  }
]
```

String functions: Return all products with "iPhone" in the name.

[http://localhost:50875/v1/Products/Product/allproducts?\\$filter=substringof\('iPhone',ProductName\)](http://localhost:50875/v1/Products/Product/allproducts?$filter=substringof('iPhone',ProductName))

```
{
  "ProductId": 4,
  "ProductName": "IPhone"
},
{
  "ProductId": 10,
  "ProductName": "IPhone 6"
},
{
  "ProductId": 11,
  "ProductName": "IPhone 6S"
}
```

Filter option could also be applied on date fields as well.

\$orderby

Let us try orderby query with the same endpoint.

Return all products with sorting on product name descending

[http://localhost:50875/v1/Products/Product/allproducts?\\$orderby=ProductName desc](http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductName desc)

Output:

```
[
  {
    "ProductId":6,
    "ProductName":"Watch"
  },
  {
    "ProductId":8,
    "ProductName":"Titan Watch"
  },
  {
    "ProductId":9,
    "ProductName":"Laptop Bag"
  },
  {
    "ProductId":1,
    "ProductName":"Laptop"
  },
  {
```



```
"ProductId":11,  
"ProductName":"iPhone 6S"  
},  
{  
"ProductId":10,  
"ProductName":"iPhone 6"  
},  
{  
"ProductId":4,  
"ProductName":"iPhone"  
},  
{  
"ProductId":12,  
"ProductName":"HP Laptop"  
},  
{  
"ProductId":2,  
"ProductName":"computer"  
},  
{  
"ProductId":5,  
"ProductName":"Bag"  
}
```



```
]
```

Return all products with sorting on product name ascending

`http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductName asc`

Output:

```
[  
  {  
    "ProductId": 5,  
    "ProductName": "Bag"  
  },  
  {  
    "ProductId": 2,  
    "ProductName": "computer"  
  },  
  {  
    "ProductId": 12,  
    "ProductName": "HP Laptop"  
  },  
  {  
    "ProductId": 4,
```



```
"ProductName": "IPhone"
},
{
  "ProductId": 10,
  "ProductName": "IPhone 6"
},
{
  "ProductId": 11,
  "ProductName": "IPhone 6S"
},
{
  "ProductId": 1,
  "ProductName": "Laptop"
},
{
  "ProductId": 9,
  "ProductName": "Laptop Bag"
},
{
  "ProductId": 8,
  "ProductName": "Titan Watch"
},
{
```



```
"ProductId": 6,  
"ProductName": "Watch"  
}  
]
```

Return all products with sorting on product id descending

[http://localhost:50875/v1/Products/Product/allproducts?\\$orderby=ProductId desc](http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductId desc)

Output:

```
[  
  {  
    "ProductId": 12,  
    "ProductName": "HP Laptop"  
  },  
  {  
    "ProductId": 11,  
    "ProductName": "iPhone 6S"  
  },  
  {  
    "ProductId": 10,  
    "ProductName": "iPhone 6"  }  
]
```



```
},  
  
{  
  "ProductId": 9,  
  "ProductName": "Laptop Bag"  
},  
  
{  
  "ProductId": 8,  
  "ProductName": "Titan Watch"  
},  
  
{  
  "ProductId": 6,  
  "ProductName": "Watch"  
},  
  
{  
  "ProductId": 5,  
  "ProductName": "Bag"  
},  
  
{  
  "ProductId": 4,  
  "ProductName": "iPhone"  
},  
  
{  
  "ProductId": 2,
```



```
"ProductName": "computer"
},
{
  "ProductId": 1,
  "ProductName": "Laptop"
}
]
```

Return all products with sorting on product id ascending

[http://localhost:50875/v1/Products/Product/allproducts?\\$orderby=ProductId asc](http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductId asc)

Output:

```
[
  {
    "ProductId": 1,
    "ProductName": "Laptop"
  },
  {
    "ProductId": 2,
    "ProductName": "computer"
  },
]
```



```
{  
  "ProductId": 4,  
  "ProductName": "IPhone"  
},  
  
{  
  "ProductId": 5,  
  "ProductName": "Bag"  
},  
  
{  
  "ProductId": 6,  
  "ProductName": "Watch"  
},  
  
{  
  "ProductId": 8,  
  "ProductName": "Titan Watch"  
},  
  
{  
  "ProductId": 9,  
  "ProductName": "Laptop Bag"  
},  
  
{  
  "ProductId": 10,  
  "ProductName": "IPhone 6"
```



```
},  
  
{  
  "ProductId": 11,  
  "ProductName": "iPhone 6S"  
},  
  
{  
  "ProductId": 12,  
  "ProductName": "HP Laptop"  
}  
]
```

\$orderby with \$top

You can make use of multiple query options to fetch the desired records. Suppose I need to fetch only 5 records from top order by ProductId ascending. To achieve this I can write the following query.

[http://localhost:50875/v1/Products/Product/allproducts?\\$orderby=ProductId asc&\\$top=5](http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductId asc&$top=5)

Output:

```
[  
  
  {
```



```
"ProductId": 1,  
  "ProductName": "Laptop"  
},  
{  
  "ProductId": 2,  
  "ProductName": "computer"  
},  
{  
  "ProductId": 4,  
  "ProductName": "IPhone"  
},  
{  
  "ProductId": 5,  
  "ProductName": "Bag"  
},  
{  
  "ProductId": 6,  
  "ProductName": "Watch"  
}  
]
```

The above output fetches 5 records with sorted ProductId.

\$skip

As the name suggests, the skip query option is used to skip the record. Let's consider following scenarios.

Select top 5 and skip 3

[http://localhost:50875/v1/Products/Product/allproducts?\\$top=5&\\$skip=3](http://localhost:50875/v1/Products/Product/allproducts?$top=5&$skip=3)

Output

```
[  
  {  
    "ProductId": 5,  
    "ProductName": "Bag"  
  },  
  {  
    "ProductId": 6,  
    "ProductName": "Watch"  
  },  
  {  
    "ProductId": 8,  
    "ProductName": "Titan Watch"  }  
]
```



```
},  
  
{  
  "ProductId": 9,  
  "ProductName": "Laptop Bag"  
},  
  
{  
  "ProductId": 10,  
  "ProductName": "iPhone 6"  
}  
]
```

\$skip with \$orderby

Order by ProductName ascending and skip 6

[http://localhost:50875/v1/Products/Product/allproducts?\\$orderby=ProductName asc &\\$skip=6](http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductName asc &$skip=6)

Output

```
[  
  
{  
  "ProductId": 1,  
  "ProductName": "Laptop"
```



```
},  
  
{  
  "ProductId": 9,  
  "ProductName": "Laptop Bag"  
},  
  
{  
  "ProductId": 8,  
  "ProductName": "Titan Watch"  
},  
  
{  
  "ProductId": 6,  
  "ProductName": "Watch"  
}  
]
```

Following are some standard filter operators and query functions you can use to create your query taken from <https://msdn.microsoft.com/en-us/library/gg334767.aspx>

Standard filter operators

The Web API supports the standard OData filter operators listed in the following table.

Operator	Description	Example
Comparison Operators		
eq	Equal	\$filter=revenue eq 100000
ne	Not Equal	\$filter=revenue ne 100000
gt	Greater than	\$filter=revenue gt 100000
ge	Greater than or equal	\$filter=revenue ge 100000
lt	Less than	\$filter=revenue lt 100000
le	Less than or equal	\$filter=revenue le 100000
Logical Operators		
and	Logical and	\$filter=revenue lt 100000 and revenue gt 2000
or	Logical or	\$filter=contains(name,'(sample)') or contains(name,'test')

not	Logical negation	\$filter=not contains(name,'sample')
Grouping Operators		
()	Precedence grouping	(contains(name,'sample') or contains(name,'test')) and revenue gt 5000

Standard query functions

The web API supports these standard OData string query functions.

Function	Example
contains	\$filter=contains(name,'(sample)')
endswith	\$filter=endswith(name,'Inc.')
startswith	\$filter=startswith(name,'a')

Paging

You can create paging enabled endpoint which means, if you have a lot of data on database, and the requirement is that client needs to show the data like 10 records per

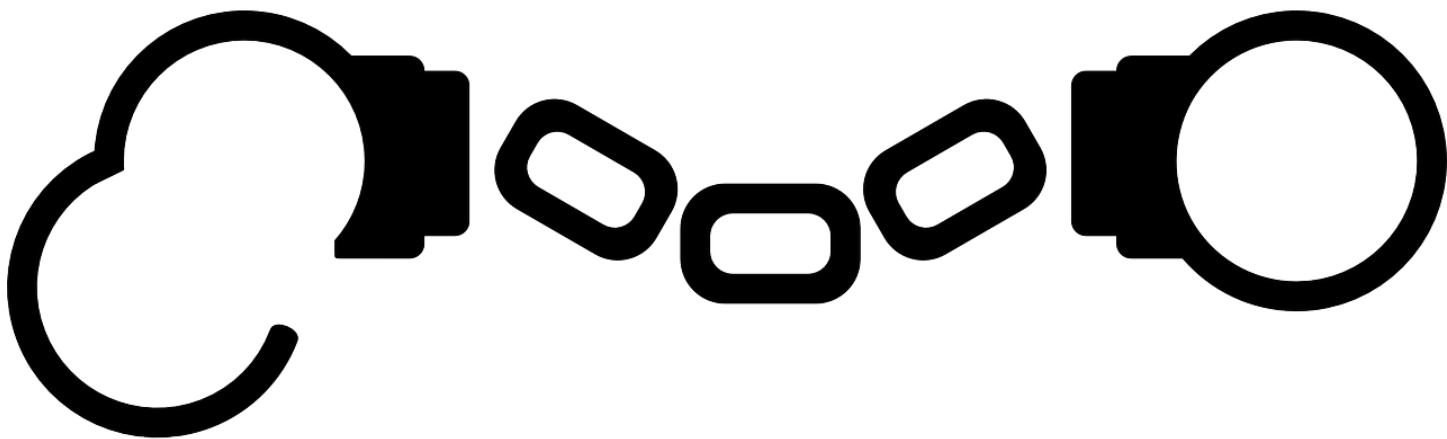
page. So it is advisable that server itself should send those 10 records per request, so that the entire data payload does not travel on network. This may also improve the performance of your services.

Let's suppose you have 10000 records on database, so you can enable your endpoint to return 10 records and entertain the request for initial record and number of records to be sent. In this case client will make request every time for next set of records fetch pagination option is used or user navigates to next page. To enable paging, just mention the page count at the [Queryable] attribute. For e.g. [Queryable(Pagesize = 10)]

So our method code becomes,

```
[Queryable(Pagesize = 10)]  
[GET("allproducts")]  
[GET("all")]  
public HttpResponseMessage Get()  
{  
    var products = _productServices.GetAllProducts().AsQueryable();  
    var productEntities = products as List<ProductEntity> ?? products.ToList();  
    if (productEntities.Any())  
        return Request.CreateResponse(HttpStatusCode.OK,  
productEntities.AsQueryable());  
    throw new ApiDataException(1000, "Products not found",  
HttpStatusCode.NotFound);  
}
```


Query Options Constraints



You can put constraints over your query options too. Suppose you do not want client to access filtering options or skip options, then at the action level you can put constraints to ignore that kind of API request. Query Option constraints are of four types,

AllowedQueryOptions

Example : [[Queryable](#)(AllowedQueryOptions =[AllowedQueryOptions.Filter](#) | [AllowedQueryOptions.OrderBy](#))]

Above example of query option states that only \$filter and \$orderby queries are allowed on the API.


```

[Queryable(AllowedQueryOptions = AllowedQueryOptions.Filter |
AllowedQueryOptions.OrderBy)]

[GET("allproducts")]

[GET("all")]

public HttpResponseMessage Get()
{
    var products = _productServices.GetAllProducts().AsQueryable();

    var productEntities = products as List<ProductEntity> ?? products.ToList();

    if (productEntities.Any())

        return Request.CreateResponse(HttpStatusCode.OK,
productEntities.AsQueryable());

    throw new ApiDataException(1000, "Products not found",
HttpStatusCode.NotFound);
}


```

So when I invoked the endpoint with \$stop query,

[http://localhost:50875/v1/Products/Product/allproducts?\\$stop=10](http://localhost:50875/v1/Products/Product/allproducts?$stop=10)

I got the following response,


```
{
  "Message": "The query specified in the URI is not valid.",
  "ExceptionMessage": "Query option 'Top' is not allowed. To allow it,
set the 'AllowedQueryOptions' property on QueryableAttribute or
QueryValidationSettings.",
  "ExceptionType": "Microsoft.Data.OData.ODataException",
  "StackTrace": "    at
System.Web.Http.OData.Query.Validators.ODataQueryValidator.Validate
QueryOptionAllowed(AllowedQueryOptions queryOption,
```



It says,

```
"Message": "The query specified in the URI is not valid.",

"ExceptionMessage": "Query option 'Top' is not allowed. To allow it, set the
'AllowedQueryOptions' property on QueryableAttribute or QueryValidationSettings."
```

That means it is not allowing other kind of queryoptions to work on this API endpoint.

AllowedOrderByProperties

Example : `[Queryable(AllowedOrderByProperties = "ProductId")]` // supply list of columns/properties

This means that the endpoint only supports sorting on the basis of ProductId. You can specify more properties for which you want to enable sorting. So as per following code,

```
[Queryable(AllowedOrderByProperties = "ProductId")]
```



```
[GET("allproducts")]
```

```
[GET("all")]
```

```
public HttpResponseMessage Get()
```

```
{
```

```
    var products = _productServices.GetAllProducts().AsQueryable();
```

```
    var productEntities = products as List<ProductEntity> ?? products.ToList();
```

```
    if (productEntities.Any())
```

```
        return Request.CreateResponse(HttpStatusCode.OK,  
productEntities.AsQueryable());
```

```
        throw new ApiDataException(1000, "Products not found",  
HttpStatusCode.NotFound);
```

```
}
```

If I try to invoke the URL :

[http://localhost:50875/v1/Products/Product/allproducts?\\$orderby=ProductName desc](http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductName desc)

It gives error in response,


Response for GET v1/Products/Product/allproducts ✕

Status

400/Bad Request


Headers

Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 1206
Expires: -1



Body

```
{
  "Message": "The query specified in the URI is not valid.",
  "ExceptionMessage": "Order by 'ProductName' is not allowed. To allow
it, set the 'AllowedOrderByProperties' property on QueryableAttribute or
QueryValidationSettings.",
  "ExceptionType": "Microsoft.Data.OData.ODataException",
  "StackTrace": "    at
System.Web.Http.OData.Query.Validators.OrderByQueryValidator.Valida
te(OrderByQueryOption orderByOption, ODataValidationSettings
validationSettings)\r\n    at
```



Says,

"Message": "The query specified in the URI is not valid.",

"ExceptionMessage": "Order by 'ProductName' is not allowed. To allow it, set the 'AllowedOrderByProperties' property on QueryableAttribute or QueryValidationSettings."

The URL: `http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductId desc` will work fine.

AllowedLogicalOperators

Example : [[Queryable](#)(AllowedLogicalOperators = [AllowedLogicalOperators](#).GreaterThan)]

In the above mentioned example, the statement states that only greaterThan i.e. “gt” logical operator is allowed in the query and query options with any other logical operator other than “gt” will return error. You can try it in your application.

AllowedArithmeticOperators

Example : [[Queryable](#)(AllowedArithmeticOperators = [AllowedArithmeticOperators](#).Add)]

In the above mentioned example, the statement states that only Add arithmetic operator is allowed while API call. You can try it in your application.

Conclusion



There are lot more things in OData that I cannot cover in one go. The purpose was to give an idea of what we can achieve using OData. You can explore more options and attributes and play around with REST API's. I hope by you'll be able to create a basic WebAPI application with all the required functionalities. The code base attached with all the articles in the series serves as a boilerplate for creating any Enterprise level WebAPI application. Keep exploring REST. Happy coding😊. Download complete source code from [GitHub](#).

References

<http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/supporting-odata-query-options>

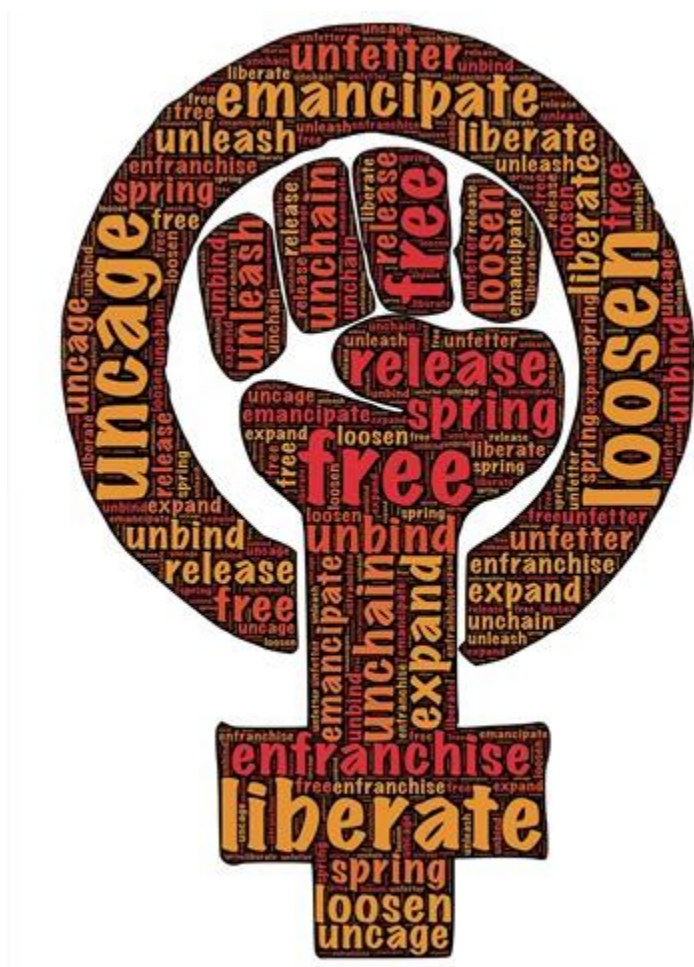
<https://msdn.microsoft.com/en-us/library/azure/gg312156.aspx>

Creating Self Hosted ASP.NET Web API with CRUD Operations in Visual Studio 2010

I have been writing a lot over WebAPIs in my Learning WebAPI series, but one crucial topic that I missed was hosting an asp.net WebAPI .

Hosting a WebAPI in IIS is pretty straight forward and is more similar to how you host a typical asp.net web application. In this article I'll explain how we can host a WebAPI in another process independent of IIS.

I'll explain how to quickly create a WebAPI having CRUD operations with Entity Framework 4.0 and then host it in an independent server. I'll call the service end points through a console application acting as a client. You can use any client to check the service end points and verify their functionality. I'll try to explain the topic with practical implementations , create a service and a test client in Visual Studio 2010 around target framework as .Net Framework 4.0.



WebAPI project

The traditional way of creating an asp.net REST service is to select a WebAPI project from visual studio , create a controller, expose endpoints and host that on IIS. But when it comes to creating a self hosted web api on windows, we need to take a windows or a console based application, that generates an exe when compiled which in turn can be used for hosting through a command prompt. You can host WebAPI 2 with Owin and that is very straight forward with the help of two nuget packages mentioned below,

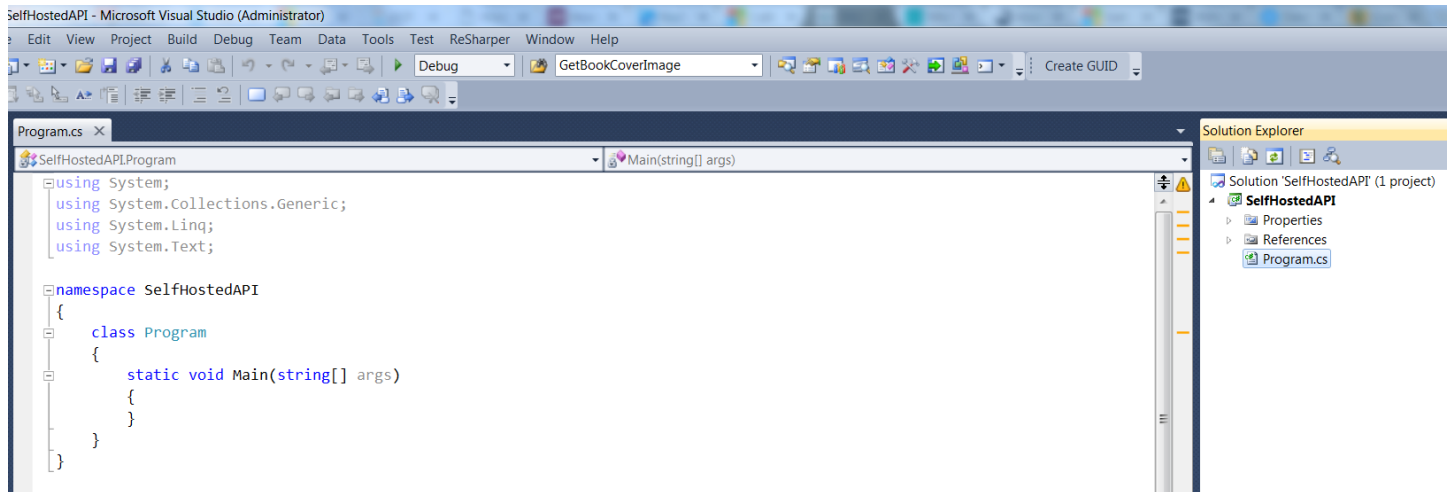
Microsoft.Owin.SelfHost

Microsoft.AspNet.WebApi.OwinSelfHost

But we'll do hosting for WebAPI 1 and will not make use of Owin for this application.

Step 1 : Create Console Application.

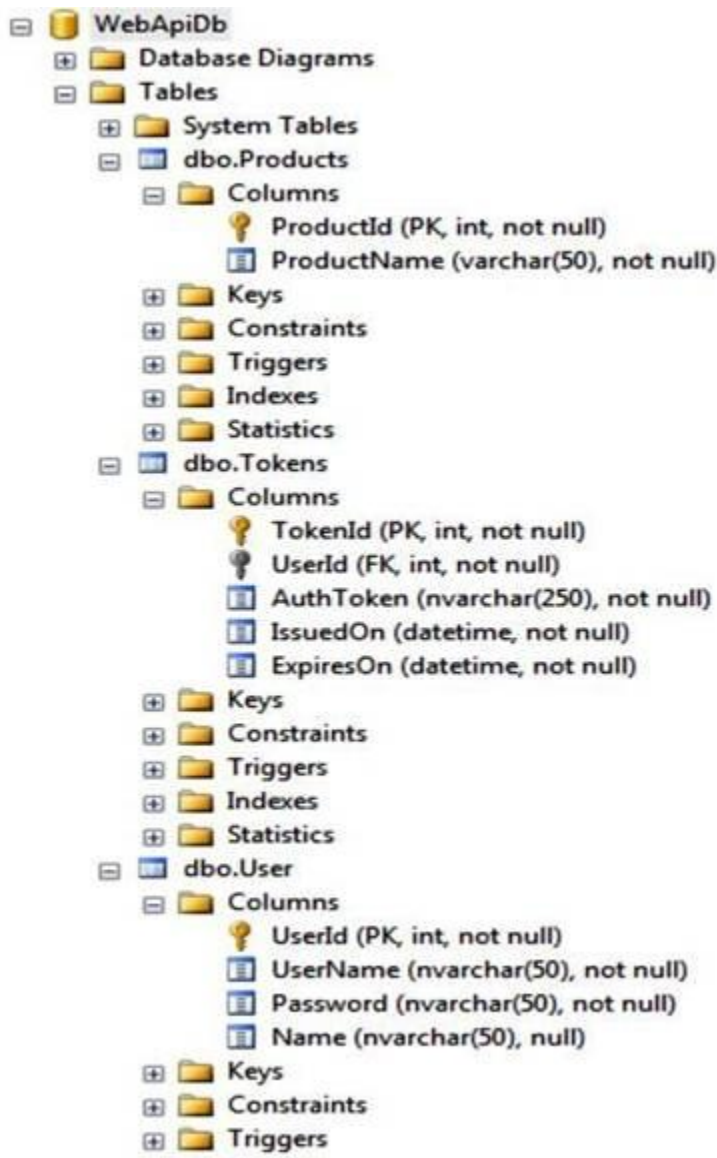
Open your visual studio and create a new console application named SelfHostedAPI



We'll add a WebAPI controller to it and write code for CRUD operations, but before that we need database and a communication for performing database transactions. I'll use EntityFramework for database transactions.

Step 2 : Create Database

You can create any database you want. I am using SQL Server and will create a database named WebAPIdb having a table named Products for performing CRUD operations. There are more tables in this database but we'll not use them.



I'll provide that database script along with this article. Script for Products table is as follows,

```

1:
2: USE [WebApiDb]
3: GO
4: /***** Object: Table [dbo].[Products] Script Date:
5: 04/14/2016 11:02:51 *****/
6: SET ANSI_NULLS ON
7: GO
8: SET QUOTED_IDENTIFIER ON
9: GO
10: SET ANSI_PADDING ON

```



```
11: GO

12: CREATE TABLE [dbo].[Products] (

13:     [ProductId] [int]

14:     IDENTITY(1,1) NOT NULL,

15:     [ProductName] [varchar](50) NOT

16:     NULL,

17: CONSTRAINT [PK_Products] PRIMARY KEY CLUSTERED

18:

19: (

20:     [ProductId] ASC

21: ) WITH (PAD_INDEX = OFF,

22: STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS

23: = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]

24: ) ON [PRIMARY]

25: GO

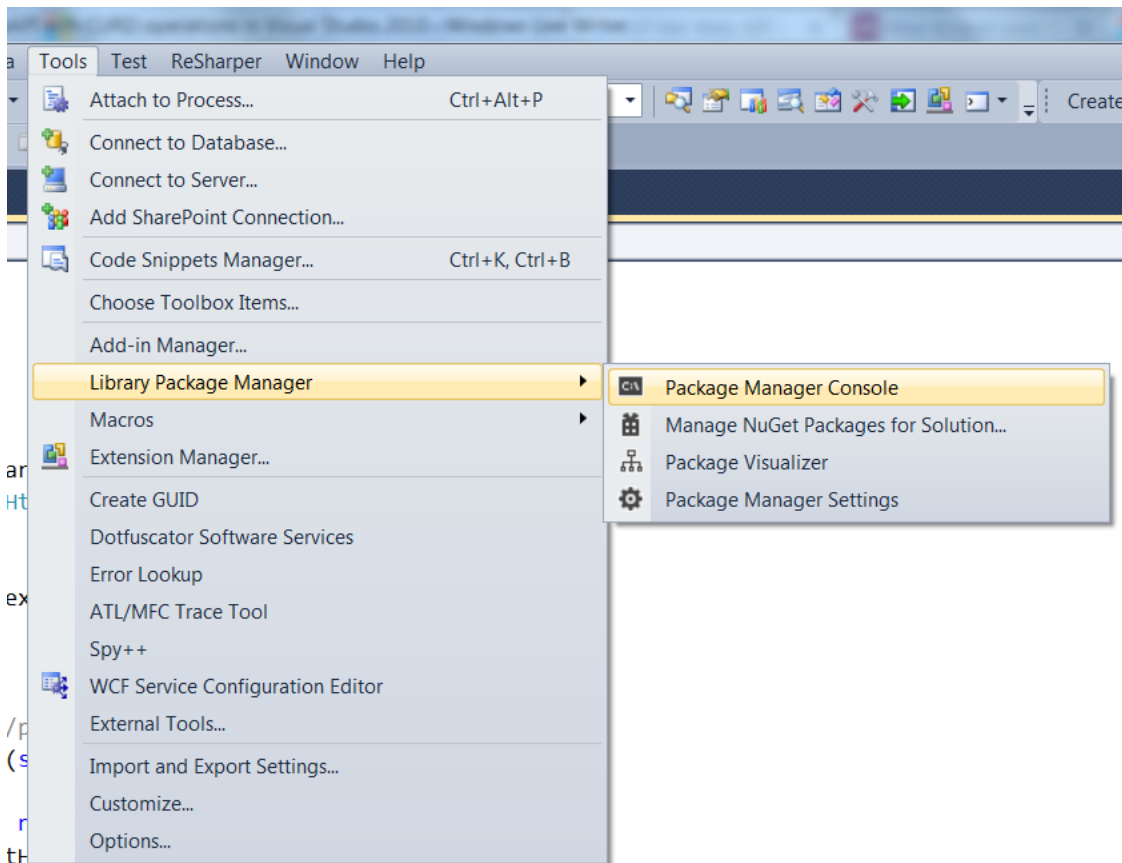
26: SET ANSI_PADDING OFF

27: GO
```

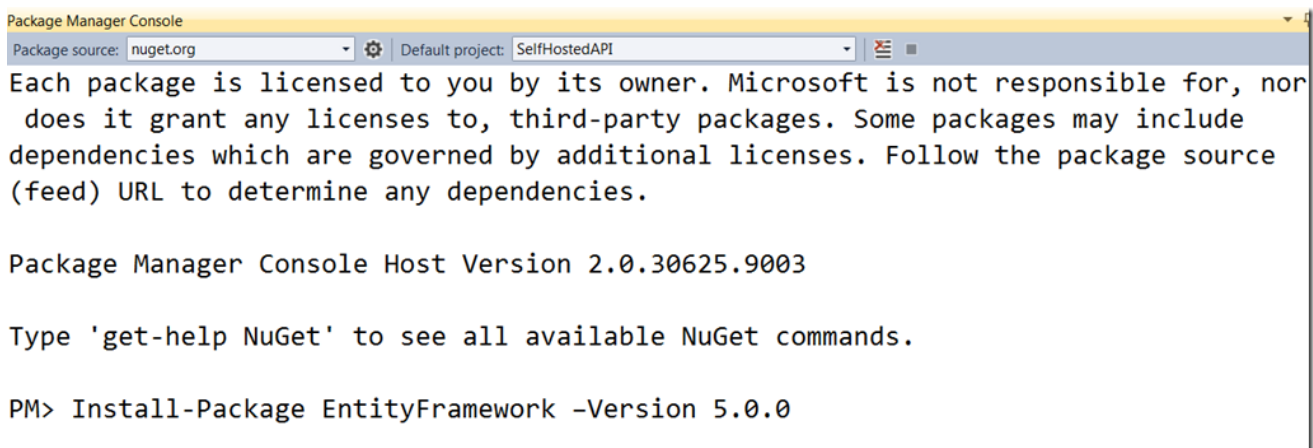
So we are done with database creation , now let us setup the entity layer for communication.

Step 2 : Set up data access using Entity Framework

In your Visual Studio, select Tool->Packet Manager->Packet Manager Console to add Entity framework package,

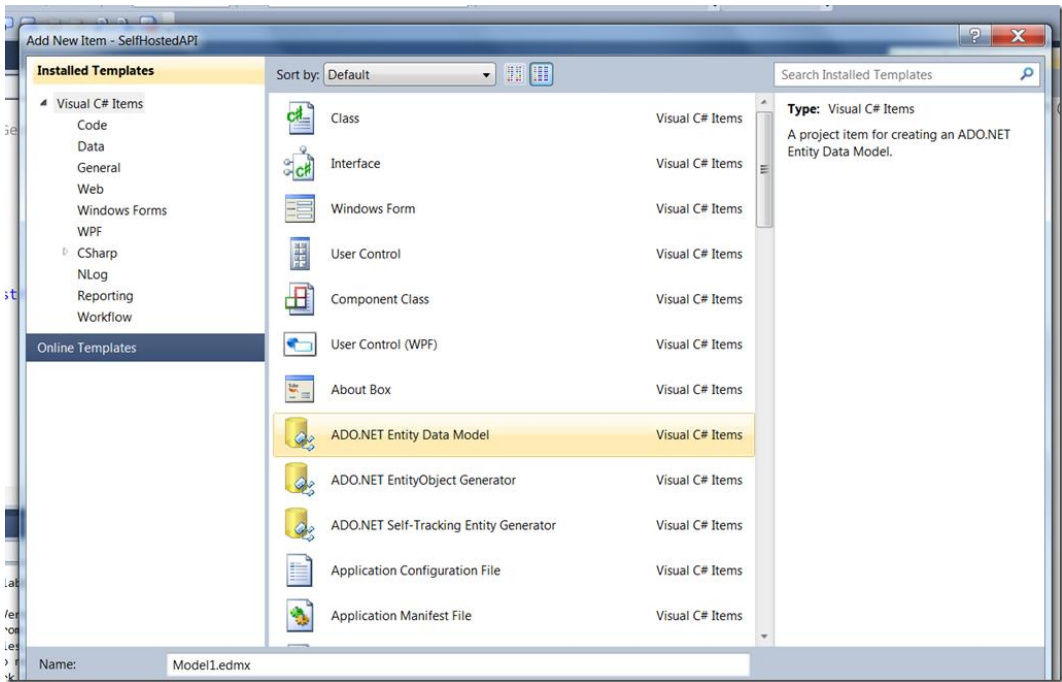


I'll install Entity Framework 5.0, as it works well with .Net Framework 4.0. So select SelfHostedAPI as Default project and type command **Install-Package EntityFramework –Version 5.0.0** and press enter.

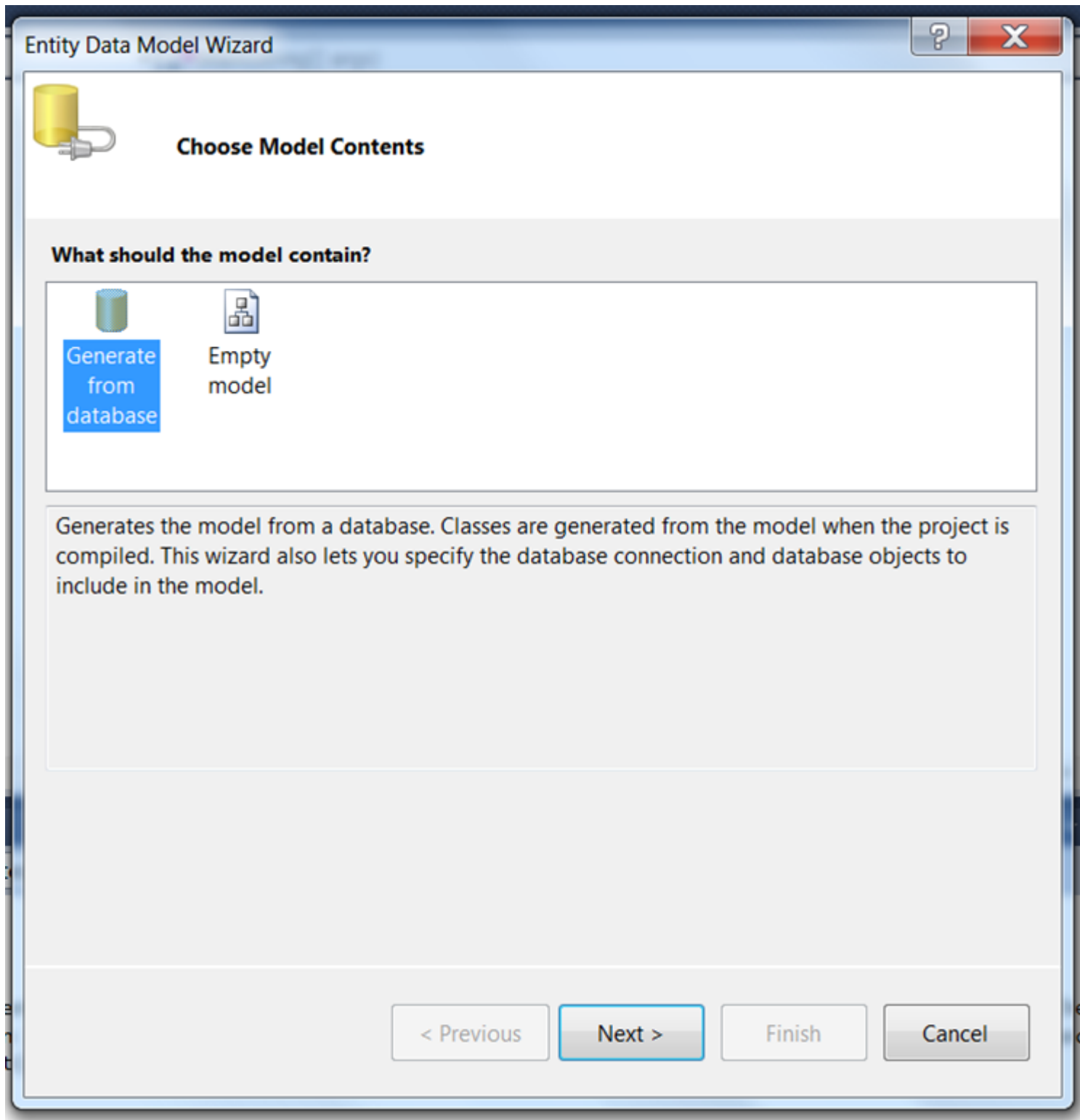


Once installed successfully, you'll see Entity framework dll added to your project.

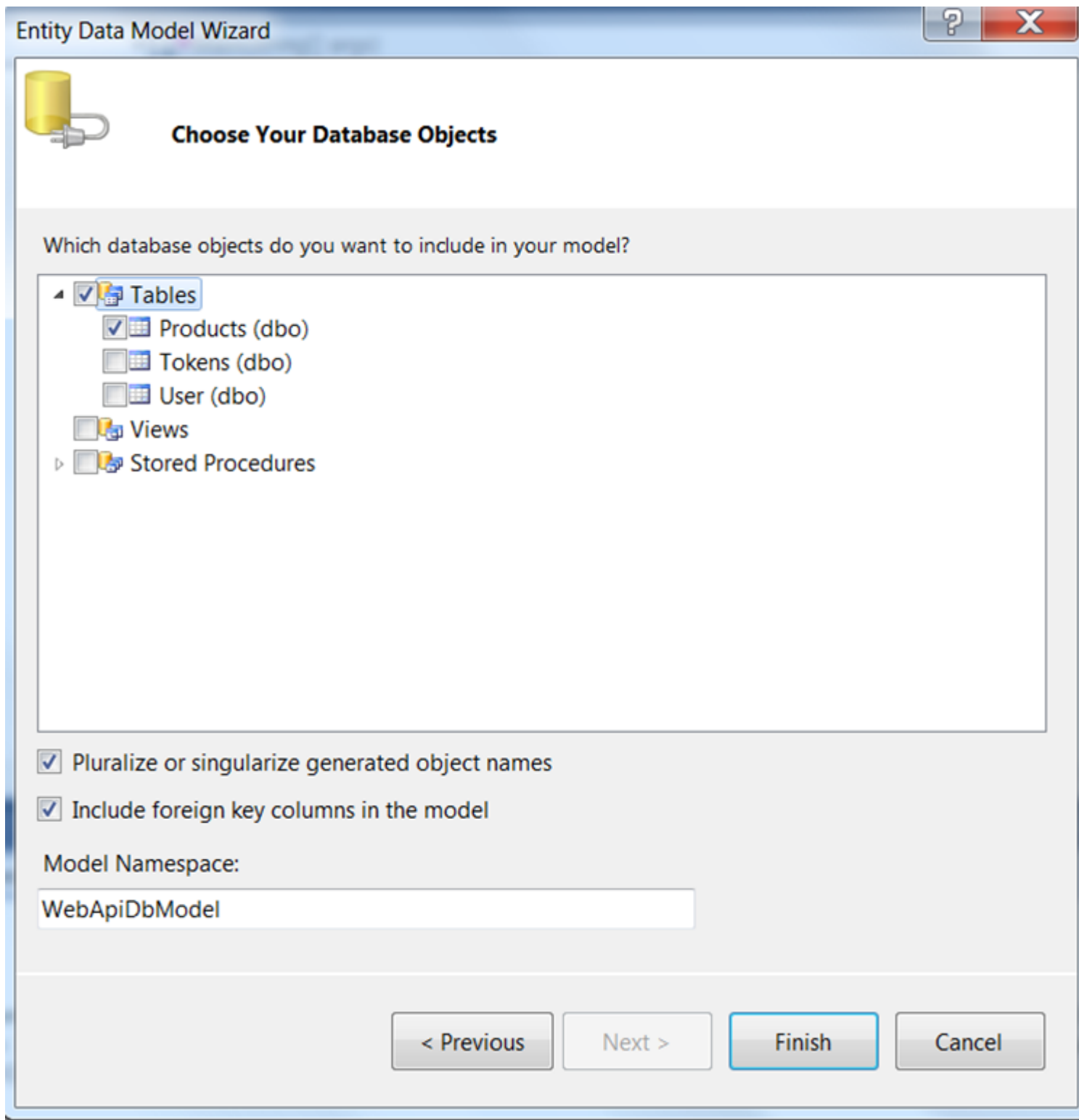
Now right click your project and add new item. Select ADO.Net Entity Data Model from list of Visual c# items.



You'll be prompted with options to generate model. Choose Generate From Database option and proceed.



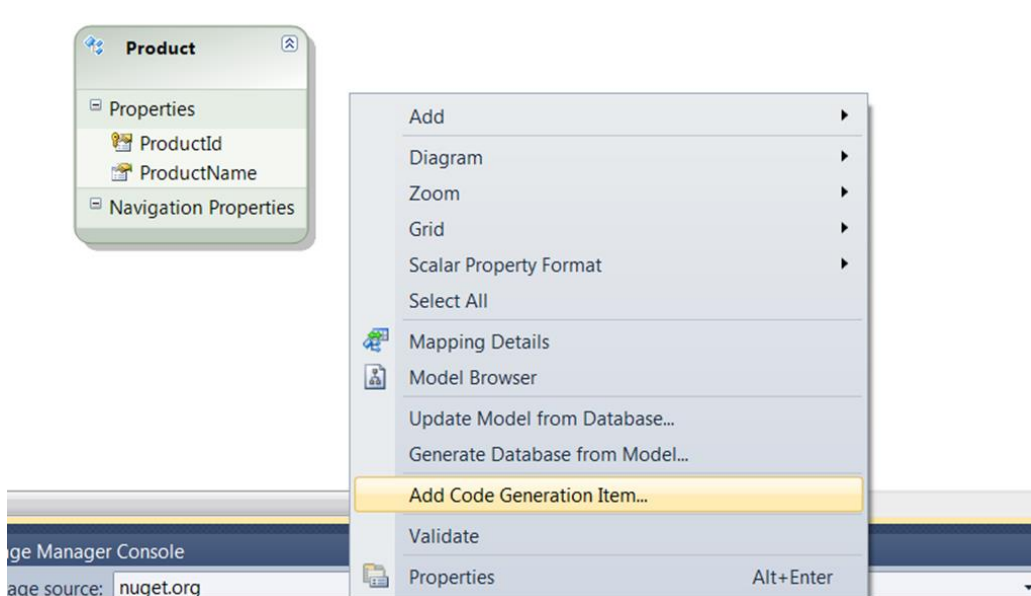
After providing database details in the next step, choose database tables that you want to map with the model. I have only selected products table as we'll perform CRUD over this table only.



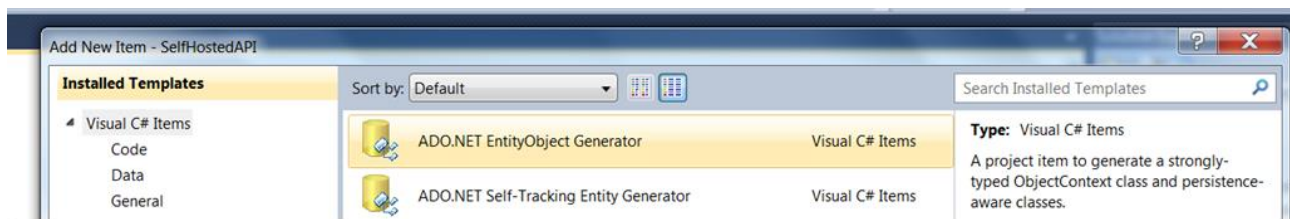
Click on Finish and your Entity Data Model will be ready with your database tables mapping. You'll see that an App.Config file is generated and a connection string specific to the selected database is added to that config file.

Now we need to generate object context that handles transactions and objectset acting as model classes mapped to the table.

Right click on your edmx view and in the context menu , click Add Code Generation Item.



In the open window for list of items select ADO.NET EntityObject Generator like shown in below image.

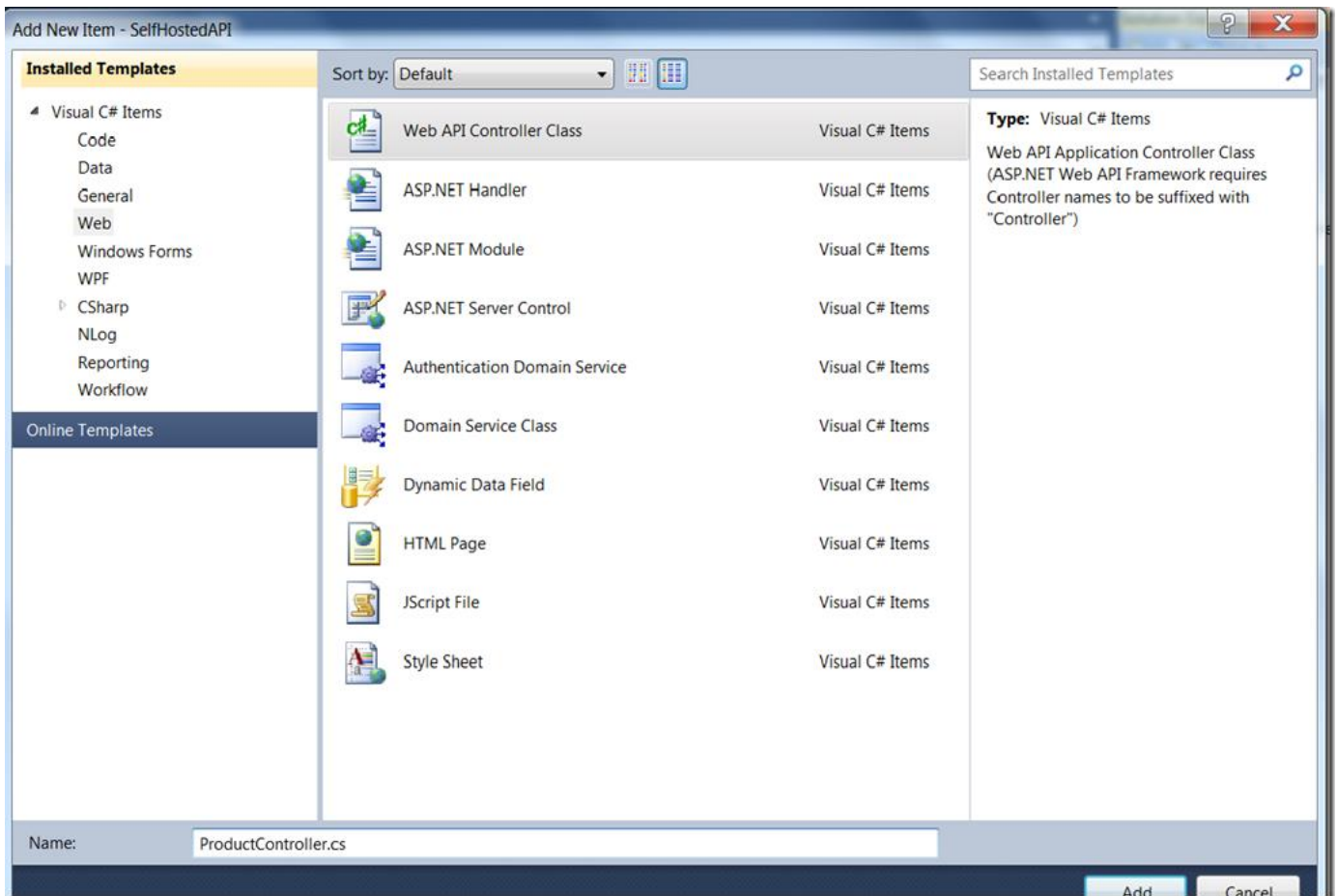


Select this and press OK, this will generate your Model1.tt class containing context and entities in the same class. Now we are done with all database related stuff. I'll now create a WebAPI controller and in place all CRUD operations in it.

Step 3: Add WebAPI Controller

Since we need an api where we can perform all CRUD operations on our database, we have to add a WebAPI controller class in the project. Since this is a console application and not a WebAPI project, so we don't have a proper structure defined for adding controller. you can create your own folder structure for the sake of understanding. I am directly adding an API controller class named ProductController in the project.

Right click on project, add new item and select WebAPI controller class from the list of items. You can name it as per your choice. I have named it as ProductController



The generated class is derived from `ApiController` class that means it is a valid WebAPI controller class. The class by default contains default CRUD methods that have following implementations,

```

1: using System;

2: using System.Collections.Generic;

3: using System.Linq;

4: using System.Net;

5: using System.Net.Http;

6: using System.Web.Http;

7:

8: namespace SelfHostedAPI

9: {

10:     public class ProductController : ApiController

11:     {

12:         // GET api/<controller>

13:         public IEnumerable<string> Get()

```



```
14:         {
15:             return new string[] { "value1", "value2" };
16:         }
17:
18:         // GET api/<controller>/5
19:         public string Get(int id)
20:         {
21:             return "value";
22:         }
23:
24:         // POST api/<controller>
25:         public void Post([FromBody]string value)
26:         {
27:         }
28:
29:         // PUT api/<controller>/5
30:         public void Put(int id, [FromBody]string value)
31:         {
32:         }
33:
34:         // DELETE api/<controller>/5
35:         public void Delete(int id)
36:         {
37:         }
38:     }
39: }
```

We'll make use of these default methods but write our own business logic for DB operations.

Step 4: Add CURD methods

We'll add all the four methods for Create, Update, Read and Delete. Note that we'll not make use of any design pattern like UnitOfWork or Repository for data persistence as our main target is to self host this service and expose its CURD endpoint.

1. Fetch All Records

Modify Get() method to return list of Product entities and make use of WebAPIEntities class generated in Model1.cs file to get list of products. The method name here signifies the type of method as well, so basically this is a Get method of the service and should be called as method type get from the client as well. Same applies to every method we write here in this class for all CURD operations.

```

1: // GET api/<controller>
2:     public IEnumerable<Product> Get()
3:     {
4:         var entities=new WebApiDbEntities();
5:         return entities.Products;
6:     }

```

in the above mentioned code base we return IEnumerable of product entities and use object of WebApiDbEntities (auto generated context class) to fetch all the objects using entities.Products.

2. Fetch product by id

Modify Get(int id) method to return a product entity. The method takes an id and returns the product specific to that id. you can enhance the method with validations and checks to make it more robust, but for the sake of understanding the concept, I am just doing it straight away.

```

1: // GET api/<controller>/5
2:     public Product Get(int id)
3:     {
4:         var entities = new WebApiDbEntities();
5:         return entities.Products.FirstOrDefault(p=>p.ProductId==id);
6:     }

```

3. Create product

```

1: // POST api/<controller>
2:     public bool Post([FromBody]Product product)
3:     {

```



```

4:         var entities = new WebApiDbEntities();
5:         entities.AddToProducts(product);
6:         var i = entities.SaveChanges();
7:         return i > 0;
8:     }

```

As the name signifies, this is a Post method, that fetches a Product class object from body of the request and add that product into entities. Note that you product will be added to actual database only when you execute entities.SaveChanges(). This method actually inserts your record in the database and returns 1 in case of successful insert else 0.

4. Edit/Update Product

```

1: // PUT api/<controller>/5
2:     public bool Put(int id, [FromBody]Product product)
3:     {
4:         using (var entities = new WebApiDbEntities())
5:         {
6:             var prod = (from p in entities.Products
7:                         where p.ProductId == id
8:                         select p).FirstOrDefault();
9:
10:            prod.ProductName = product.ProductName;
11:            var i=entities.SaveChanges();
12:            return i > 0;
13:        }
14:    }

```

Since this is an update operation, we name it as Put method, and as the name signifies, it is of PUT type. The method takes id and product object as an argument, where first an actual product from database is fetched having id that is passed as an argument and then that product is modified with the details of parameter product and then again saved to database. In our case we have only product name that could be changed because id is fixed primary key, so we update the product name of a product in this method and save changes to that entity.

5. Delete product


```

1: // DELETE api/<controller>/5
2:     public bool Delete(int id)
3:     {
4:         using (var entities = new WebApiDbEntities())
5:         {
6:             var prod = (from p in entities.Products
7:                         where p.ProductId == id
8:                         select p).FirstOrDefault();
9:
10:            entities.Products.DeleteObject(prod);
11:            var i=entities.SaveChanges();
12:            return i > 0;
13:        }
14:    }

```

The above mentioned delete method is of DELETE type and accepts id of the product as a parameter. The method fetches product from database w.r.t. passed id and then deletes that product and save changes. the implementation is pretty simple and self explanatory.

With this method we have completed all our CRUD endpoints that we needed to expose. As you can see I have not applied any special routing for endpoints and rely upon the default routing provided by WebAPI i.e. api/<controller>/<id>

Step 5: Hosting WebAPI

Here comes the most important piece of this post, “self hosting”. Remember when you created SelfHostedAPI project, it was a console application and so it came with a Program.cs file created within the project. The Program.cs file contains main method i.e. entry point of the application. We’ll use this main method to write self hosting code for our WebAPI.

Before we write any code we need to add a nuget package through Package manager console. This package contains hosting specific classes required to host API in console application i.e. independently in separate process other than IIS. Note that in WebAPI 2 we have Owin middleware that provides this flexibility.

Since we are using Visual Studio 2010 and .net framework 4.0, we need to install a specific version of package named [Microsoft.AspNet.WebApi.SelfHost](#). the compatible version that I found was version 4.0.20710

Download Package Statistics Documentation Downloads Blog

Microsoft ASP.NET Web API Self Host 4.0.20710

This package contains everything you need to host ASP.NET Web API within your own process (outside of IIS). ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework.

To install Microsoft ASP.NET Web API Self Host, run the following command in the [Package Manager Console](#)

```
PM> Install-Package Microsoft.AspNet.WebApi.SelfHost -Version 4.0.20710
```

[Tweet](#)

So open your package manager console and choose default project and execute the command **“Install-Package Microsoft.AspNet.WebApi.SelfHost -Version 4.0.20710”**

This installs the package for your project and now you can use it for implementation.

Open the Program.cs file and add a namespace

```
1: using System.Web.Http.SelfHost;
```

and in the main method define the base address of your endpoint that you want to expose, I am choosing the endpoint to be 8082. make sure you are running your visual studio in Administrator mode else you’ll have to change few configurations to work it for you. I have taken the following explanation from this [asp.net article](#) to explain configuration changes,

“Add an HTTP URL Namespace Reservation

This application listens to http://localhost:8080/. By default, listening at a particular HTTP address requires administrator privileges. When you run the tutorial, therefore, you may get this error: "HTTP could not register URL http://+:8080/" There are two ways to avoid this error:

- *Run Visual Studio with elevated administrator permissions, or*
- *Use Netsh.exe to give your account permissions to reserve the URL.*

To use Netsh.exe, open a command prompt with administrator privileges and enter the following command:following command:

```
netsh http add urlacl url=http://+:8080/ user=machine\username
```

where machine\username is your user account.

When you are finished self-hosting, be sure to delete the reservation:

```
netsh http delete urlacl url=http://+:8080/
```

1. Define an object for SelfHostConfiguration as follows,


```
1: var config = new HttpSelfHostConfiguration("http://localhost:8082");
```

2. Define the default route of your WebAPI

```
1: config.Routes.MapHttpRoute(
2:     "API Default", "api/{controller}/{id}",
3:     new { id = RouteParameter.Optional });
```

This is the default route that our service will follow while running.

3. Start server in a process

```
1: using (var server = new HttpSelfHostServer(config))
2: {
3:     server.OpenAsync().Wait();
4:     Console.WriteLine("Server started....");
5:     Console.WriteLine("Press Enter to quit.");
6:     Console.ReadLine();
7: }
```

The above piece of code is used to host and start a server for the service that we have created. As you can see its just few lines of code to get our service started in a separate process and we don't actually need to rely upon IIS server.

Hence our Program.cs becomes,

```
1: using System;
2: using System.Web.Http;
3: using System.Web.Http.SelfHost;
4:
5: namespace SelfHostedAPI
6: {
7:     class Program
8:     {
9:         static void Main(string[] args)
10:        {
11:            var config = new HttpSelfHostConfiguration("http://localhost:8082");
```



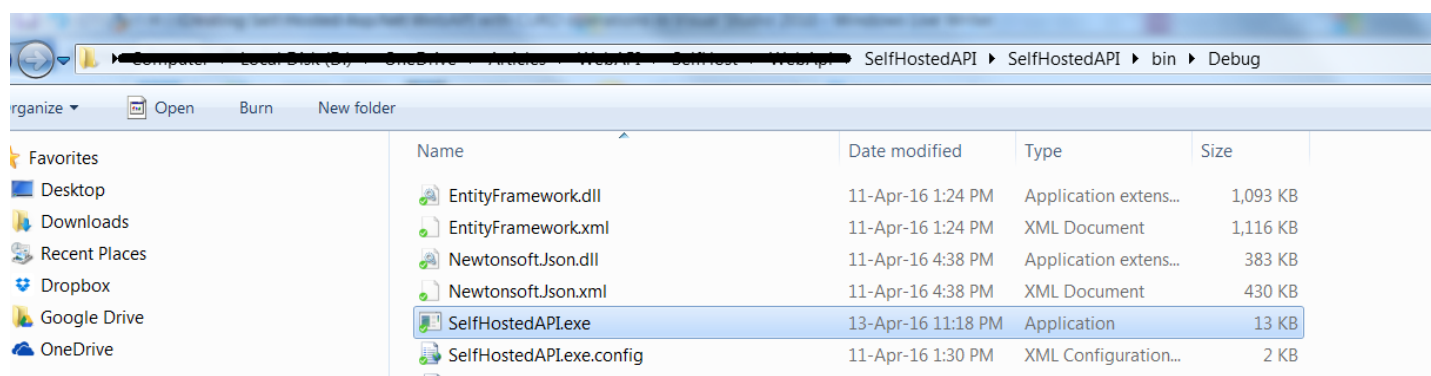
```

12:
13:         config.Routes.MapHttpRoute(
14:             "API Default", "api/{controller}/{id}",
15:             new { id = RouteParameter.Optional });
16:
17:         using (var server = new HttpSelfHostServer(config))
18:         {
19:             server.OpenAsync().Wait();
20:             Console.WriteLine("Server started....");
21:             Console.WriteLine("Press Enter to quit.");
22:             Console.ReadLine();
23:         }
24:     }
25: }
26: }

```

Now when you start the application by pressing F5, you'll get your server started and service endpoints listening to your request. We'll test the end points with our own test client that we are about to create. but before that let us start our server.

Compile the application and in windows explorer navigate to SelfHostedAPI.exe in bin\debug folder and run it as an administrator.



Your server will start immediately,


```
Administrator: Windows Command Processor - SelfHostedAPI.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>d:
D:\>cd E:\Code\1\ASP.NET\SelfHostedAPI\bin\Debug
D:\Code\1\ASP.NET\SelfHostedAPI\bin\Debug>SelfHostedAPI.exe
Server started....
Press Enter to quit.
```

And when you type the URL in browser <http://localhost:8082>, you'll see that the server actually returns a response of resource not found,

```
<Error>
  <Message>
    No HTTP resource was found that matches the request URI 'http://localhost:8082/'.
  </Message>
  <MessageDetail>No route data was found for this request.</MessageDetail>
</Error>
```

That means our port is listening to the requests.

WebAPI Test Client

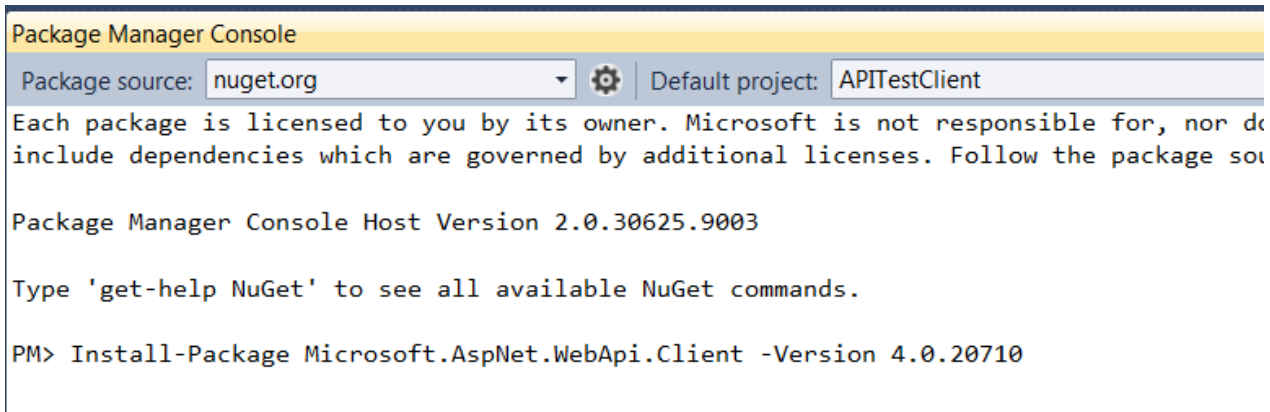
Now that we know our services are up and running, its time to test them through a test client. You can use your own test client or build a one as a console application. I am building a separate test client in .net itself to test the services.

Step1: Add a console application

Add a console application in the same or another solution with the name APITestClient or the name of your choice. We'll use Program.cs to write all the code for calling WebAPI methods. But before that we need to install a nuget package that helps us in creating a httpclient through which we'll make api calls.

Step2: Add web client package

Open Library package manager, select APITestClient as default project and execute the command **"Install-Package Microsoft.AspNet.WebApi.Client -Version 4.0.20710"**



```

Package Manager Console
Package source: nuget.org | Default project: APITestClient
Each package is licensed to you by its owner. Microsoft is not responsible for, nor do we
include dependencies which are governed by additional licenses. Follow the package source
Package Manager Console Host Version 2.0.30625.9003
Type 'get-help NuGet' to see all available NuGet commands.
PM> Install-Package Microsoft.AspNet.WebApi.Client -Version 4.0.20710

```

This will install the necessary package and its dependencies in your test client project.

Step3: Setup client

Time to code, open program.cs and add namespace ,

```

1: using System.Net.Http;
2: using System.Net.Http.Headers;

```

Define HttpClient variable,

```

1: private static readonly HttpClient Client = new HttpClient();

```

and in Main method initialize the client with basic requirements like with base address of services to be called and media type,

```

1: Client.BaseAddress = new Uri("http://localhost:8082");
2: Client.DefaultRequestHeaders.Accept.Clear();
3: Client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));

```

All set , now we can write CRUD calling methods and call them from main method.

Step4 : Calling Methods

1. GetAllProducts :

```

1: /// <summary>
2:     /// Fetch all products
3:     /// </summary>
4:     private static void GetAllProducts()
5:     {
6:         HttpResponseMessage resp = Client.GetAsync("api/product").Result;

```



```

7:         resp.EnsureSuccessStatusCode();

8:

9:         var products =
resp.Content.ReadAsAsync<IEnumerable<SelfHostedAPI.Product>>().Result.ToList();

10:        if (products.Any())

11:        {

12:            Console.WriteLine("Displaying all the products...");

13:            foreach (var p in products)

14:            {

15:                Console.WriteLine("{0} {1} ", p.ProductId, p.ProductName);

16:            }

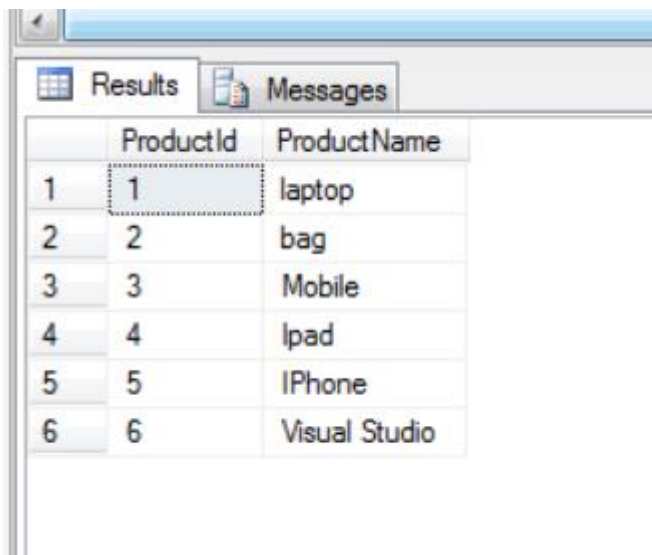
17:        }

18:    }

```

The above method makes call to “api/product “ endpoint via HttpClient instance and expects a result in HttpResponseMessage from service. Note that it uses the method GetAsync to make an endpoint call, this states that it is calling a Get method of REST API.

We have about 6 records in database that needs to be displayed,



The screenshot shows the 'Results' window in Visual Studio. It contains a table with two columns: 'ProductId' and 'ProductName'. There are six rows of data. The first row is highlighted with a dashed border.

	ProductId	ProductName
1	1	laptop
2	2	bag
3	3	Mobile
4	4	Ipad
5	5	IPhone
6	6	Visual Studio

Let’s run the application by calling the method from Main method but before that make sure your WebAPI is running in console application like we did earlier,

```

1: private static void Main(string[] args)

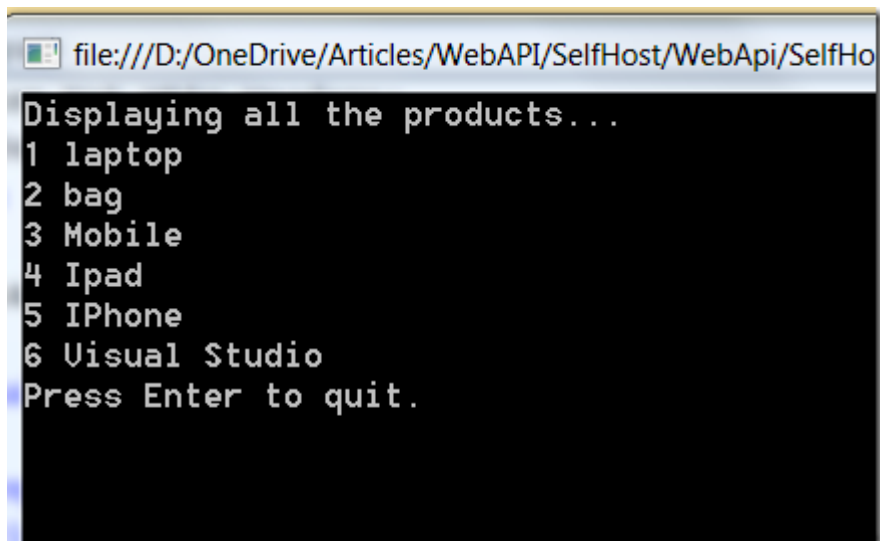
2: {

```



```
3:      Client.BaseAddress = new Uri("http://localhost:8082");  
4:      Client.DefaultRequestHeaders.Accept.Clear();  
5:      Client.DefaultRequestHeaders.Accept.Add(new  
MediaTypeWithQualityHeaderValue("application/json"));  
6:      GetAllProducts();  
7:  
8:      Console.WriteLine("Press Enter to quit.");  
9:      Console.ReadLine();  
10: }
```

Result is as below.



```
file:///D:/OneDrive/Articles/WebAPI/SelfHost/WebApi/SelfHo  
Displaying all the products...  
1 laptop  
2 bag  
3 Mobile  
4 Ipad  
5 iPhone  
6 Uisual Studio  
Press Enter to quit.
```

Hurray, we got all our products from database to this client.



This proves that our service is well hosted and working fine. We can define other methods too in a similar way and test the API endpoints.

2. GetProduct():

This method fetches product by id.

```
1: /// <summary>
2: /// Get product by id
3: /// </summary>
4: private static void GetProduct ()
5: {
6:     const int id = 1;
7:     var resp = Client.GetAsync(string.Format("api/product/{0}", id)).Result;
8:     resp.EnsureSuccessStatusCode();
9:
10:    var product = resp.Content.ReadAsAsync<SelfHostedAPI.Product>().Result;
11:    Console.WriteLine("Displaying product having id : " + id);
```

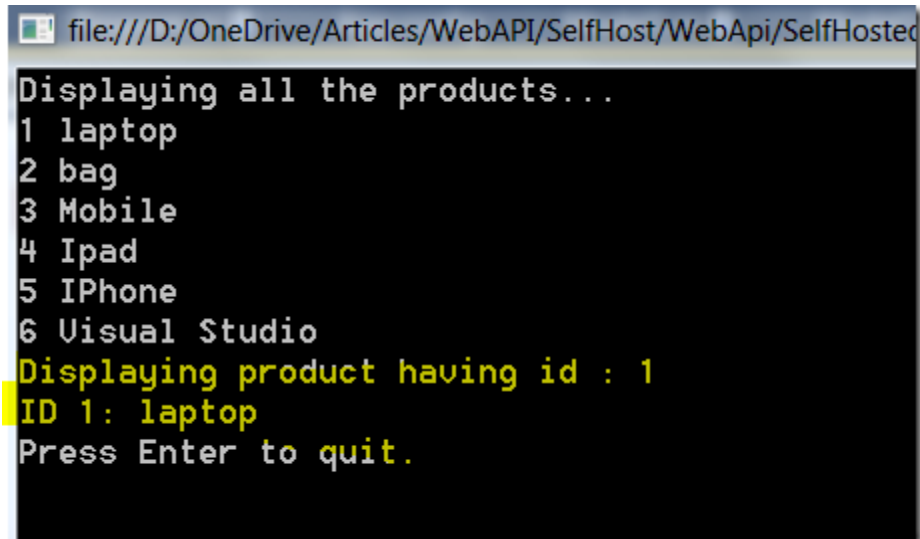


```

12:     Console.WriteLine("ID {0}: {1}", id, product.ProductName);
13: }

```

Again the method is self explanatory, I have by default called this method for product id “1”, you can customize the method as per your need. Just compile and run the method. We get,



```

file:///D:/OneDrive/Articles/WebAPI/SelfHost/WebApi/SelfHosted
Displaying all the products...
1 laptop
2 bag
3 Mobile
4 Ipad
5 IPhone
6 Uisual Studio
Displaying product having id : 1
ID 1: laptop
Press Enter to quit.

```

Hence we get the result.

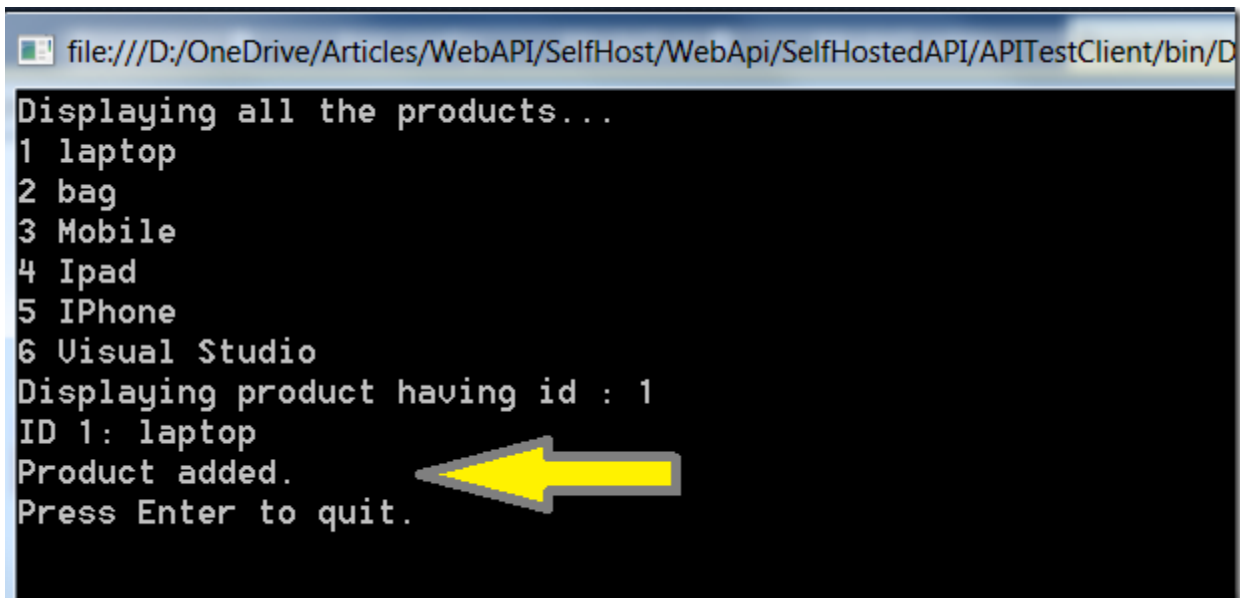
3. AddProduct():

```

1: /// <summary>
2:     /// Add product
3:     /// </summary>
4:     private static void AddProduct ()
5:     {
6:         var newProduct = new Product() { ProductName = "Samsung Phone" };
7:         var response = Client.PostAsJsonAsync("api/product", newProduct);
8:         response.Wait();
9:         if (response.Result.IsSuccessStatusCode)
10:        {
11:            Console.WriteLine("Product added.");
12:        }
13:    }

```


In above method I am trying to add a new product named “Samsung Phone” in our existing product list. Since we have auto id generation at database, so we don’t have to provide the id of the product. It will automatically get inserted in database with a unique id. Note that this method makes call with a Post type method to API end point. Run the application.



```
file:///D:/OneDrive/Articles/WebAPI/SelfHost/WebApi/SelfHostedAPI/APITestClient/bin/D
Displaying all the products...
1 laptop
2 bag
3 Mobile
4 Ipad
5 IPhone
6 Visual Studio
Displaying product having id : 1
ID 1: laptop
Product added.
Press Enter to quit.
```

It says product added. now let’s go to database and check our table. Execute a select query over your table in Sql Server database and we get one new product with id “7” added in the table having name “Samsung Phone”,


```
/* ***** Script for Selection of Products ***** */  
SELECT TOP 1000 [ProductId]  
               , [ProductName]  
FROM [WebApiDb].[dbo].[Products]
```



Results



Messages

	ProductId	ProductName
1	1	laptop
2	2	bag
3	3	Mobile
4	4	Ipad
5	5	IPhone
6	6	Visual Studio
7	7	Samsung Phone

4. EditProduct():

```
1: /// <summary>  
2:     /// Edit product  
3:     /// </summary>  
4:     private static void EditProduct()
```

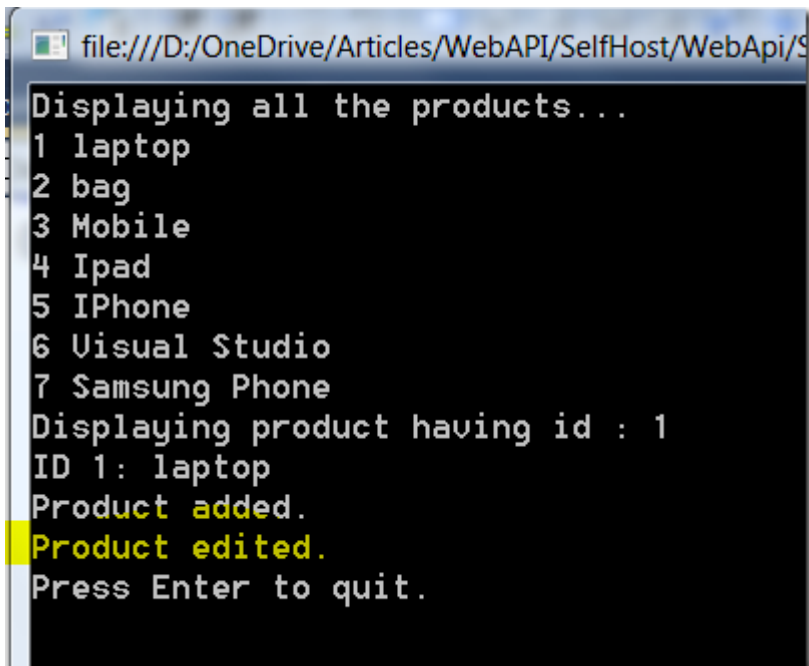


```

5:      {
6:          const int productToEdit = 4;
7:          var product = new Product() { ProductName = "Xamarin" };
8:
9:          var response =
10:              Client.PutAsJsonAsync("api/product/" + productToEdit, product);
11:          response.Wait();
12:          if (response.Result.IsSuccessStatusCode)
13:          {
14:              Console.WriteLine("Product edited.");
15:          }
16:
17:      }

```

In above code I am editing product having product id “4” and changing the existing product name i.e. iPad to Xamarin. Note that this method makes call with a Put type method to API end point. Run the application

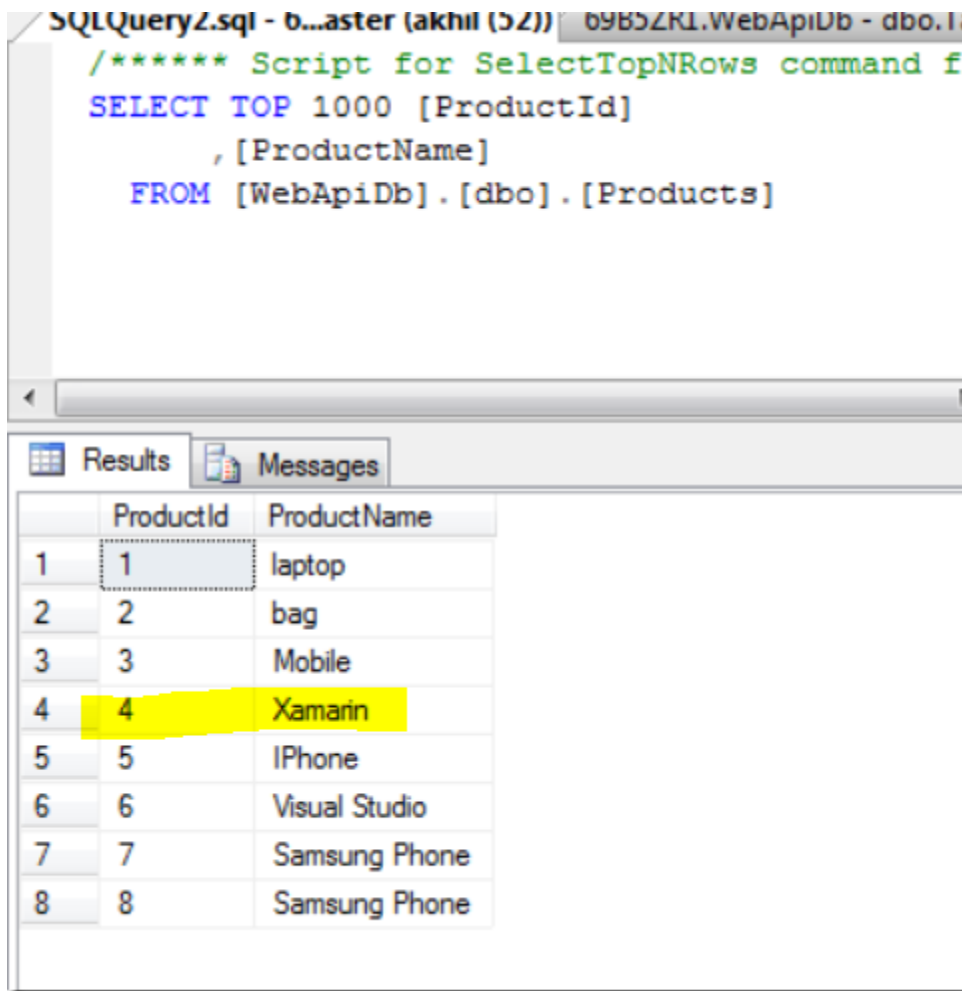


```

file:///D:/OneDrive/Articles/WebAPI/SelfHost/WebApi/S
Displaying all the products...
1 laptop
2 bag
3 Mobile
4 Ipad
5 iPhone
6 Visual Studio
7 Samsung Phone
Displaying product having id : 1
ID 1: laptop
Product added.
Product edited.
Press Enter to quit.

```

In database,



We see here that our existing product named Ipad is updated to new name “Xamarin”. We see that one new product has also been added with id 8, that’s because we again called add method from main method , ideally we would have commented it out while testing edit method 😊.

4. DeleteProduct():

```

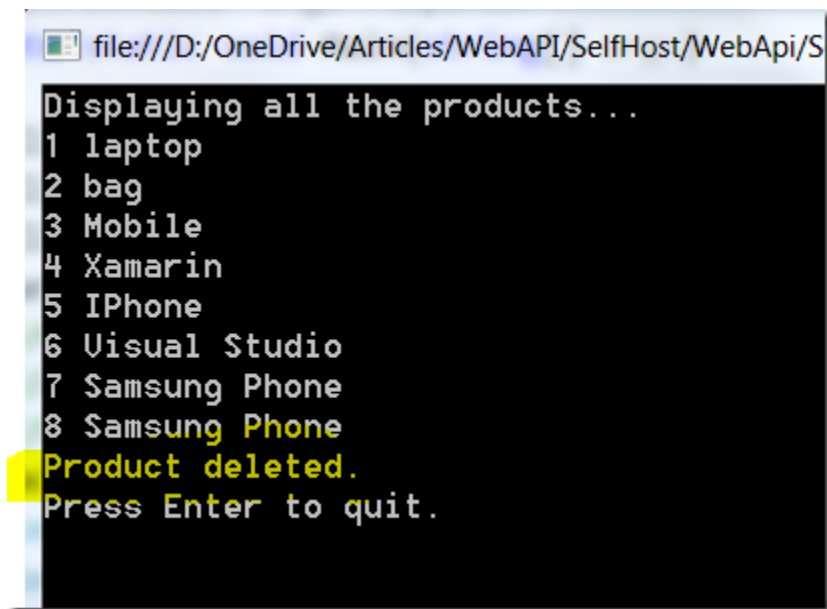
1: /// <summary>
2:     /// Delete product
3:     /// </summary>
4:     private static void DeleteProduct()
5:     {
6:         const int productToDelete = 2;
7:         var response = Client.DeleteAsync("api/product/" + productToDelete);
8:         response.Wait();
9:         if (response.Result.IsSuccessStatusCode)
10:        {

```



```
11:             Console.WriteLine("Product deleted.");  
12:         }  
13:     }
```

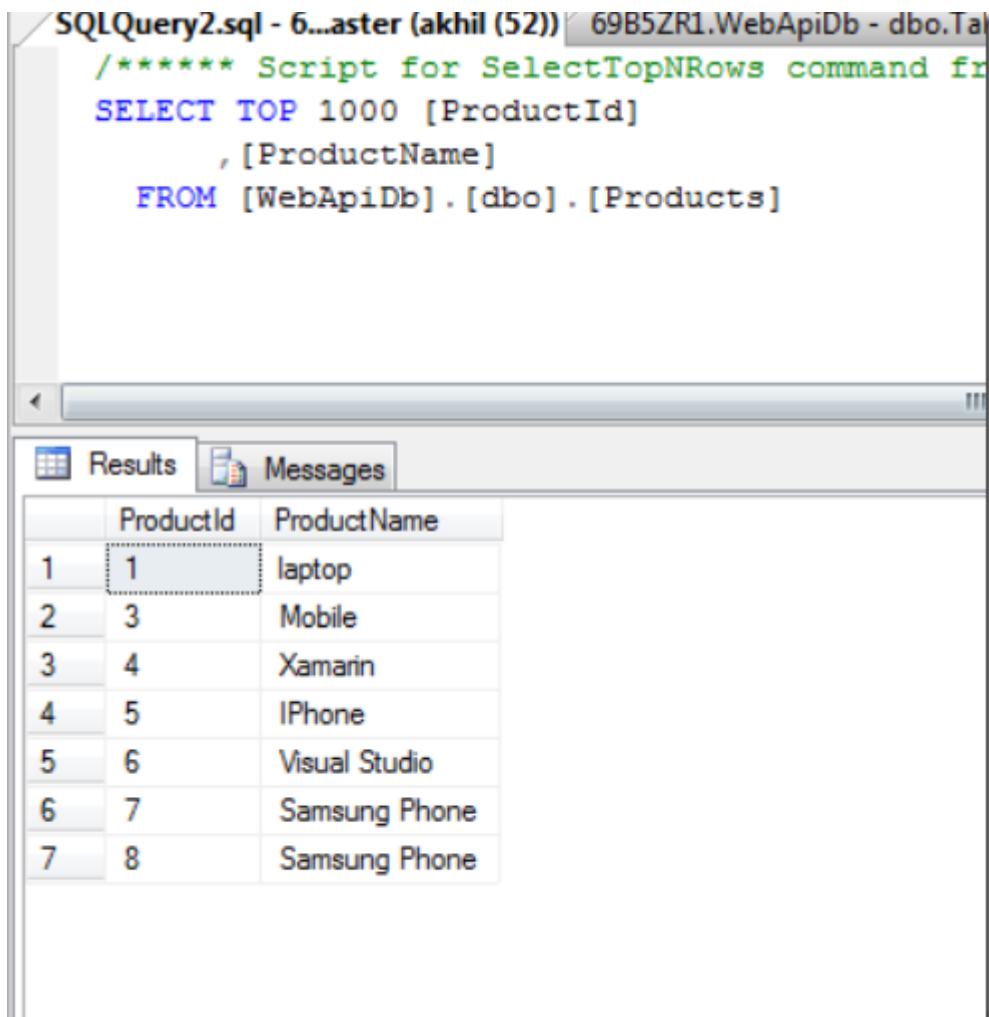
In above method we delete a product having id 2. Note that this method makes call with a Delete type method to API end point. Run the application



The screenshot shows a console window titled "file:///D:/OneDrive/Articles/WebAPI/SelfHost/WebApi/S". The output text is as follows:

```
Displaying all the products...  
1 laptop  
2 bag  
3 Mobile  
4 Xamarin  
5 iPhone  
6 Visual Studio  
7 Samsung Phone  
8 Samsung Phone  
Product deleted.  
Press Enter to quit.
```

Product deleted. Let's check in database,



We see product with id “2” as deleted. So , we have performed all the CRUD operations on a self hosted WebAPI. And the result was as expected. Following is the code for Program.cs file in consolidation.

```

1: #region Using namespaces
2: using System;
3: using System.Collections.Generic;
4: using System.Linq;
5: using System.Net.Http;
6: using System.Net.Http.Headers;
7: using SelfHostedAPI;
8: #endregion
9:
10: namespace APITestClient

```



```
11: {
12:     internal class Program
13:     {
14:         #region Private member variables
15:         private static readonly HttpClient Client = new HttpClient();
16:         #endregion
17:
18:         #region Main method for execution entry
19:         /// <summary>
20:         /// Main method
21:         /// </summary>
22:         /// <param name="args"></param>
23:         private static void Main(string[] args)
24:         {
25:             Client.BaseAddress = new Uri("http://localhost:8082");
26:             Client.DefaultRequestHeaders.Accept.Clear();
27:             Client.DefaultRequestHeaders.Accept.Add(new
MediaTyewithQualityHeaderValue("application/json"));
28:             GetAllProducts();
29:             GetProduct();
30:             AddProduct();
31:             EditProduct();
32:             DeleteProduct();
33:             Console.WriteLine("Press Enter to quit.");
34:             Console.ReadLine();
35:         }
36:         #endregion
37:
38:         #region Private client methods
```



```
39:         /// <summary>
40:         /// Fetch all products
41:         /// </summary>
42:         private static void GetAllProducts()
43:         {
44:             HttpResponseMessage resp = Client.GetAsync("api/product").Result;
45:             resp.EnsureSuccessStatusCode();
46:
47:             var products =
resp.Content.ReadAsAsync<IEnumerable<SelfHostedAPI.Product>>().Result.ToList();
48:             if (products.Any())
49:             {
50:                 Console.WriteLine("Displaying all the products...");
51:                 foreach (var p in products)
52:                 {
53:                     Console.WriteLine("{0} {1} ", p.ProductId, p.ProductName);
54:                 }
55:             }
56:         }
57:
58:         /// <summary>
59:         /// Get product by id
60:         /// </summary>
61:         private static void GetProduct()
62:         {
63:             const int id = 1;
64:             var resp = Client.GetAsync(string.Format("api/product/{0}",
id)).Result;
65:             resp.EnsureSuccessStatusCode();
66:
```



```

67:         var product =
resp.Content.ReadAsAsync<SelfHostedAPI.Product>().Result;

68:         Console.WriteLine("Displaying product having id : " + id);
69:         Console.WriteLine("ID {0}: {1}", id, product.ProductName);
70:     }
71:
72:     /// <summary>
73:     /// Add product
74:     /// </summary>
75:     private static void AddProduct()
76:     {
77:         var newProduct = new Product() { ProductName = "Samsung Phone" };
78:         var response = Client.PostAsJsonAsync("api/product", newProduct);
79:         response.Wait();
80:         if (response.Result.IsSuccessStatusCode)
81:         {
82:             Console.WriteLine("Product added.");
83:         }
84:     }
85:
86:     /// <summary>
87:     /// Edit product
88:     /// </summary>
89:     private static void EditProduct()
90:     {
91:         const int productToEdit = 4;
92:         var product = new Product() { ProductName = "Xamarin" };
93:
94:         var response =

```



```
95:             Client.PutAsJsonAsync("api/product/" + productToEdit, product);
96:         response.Wait();
97:         if (response.Result.IsSuccessStatusCode)
98:         {
99:             Console.WriteLine("Product edited.");
100:        }
101:
102:    }
103:
104:    /// <summary>
105:    /// Delete product
106:    /// </summary>
107:    private static void DeleteProduct()
108:    {
109:        const int productToDelete = 2;
110:        var response = Client.DeleteAsync("api/product/" + productToDelete);
111:        response.Wait();
112:        if (response.Result.IsSuccessStatusCode)
113:        {
114:            Console.WriteLine("Product deleted.");
115:        }
116:    }
117:
118:    #endregion
119: }
120: }
```


Conclusion

I have tried to keep this tutorial simple and straight forward, and explain each and every step through which you can create a simple WebAPI with all CRUD operations using Entity Framework, and finally self host that API and test it. I hope you enjoyed reading this article. You can download the complete source code from [github](#).

Index

A

Add Code Generation Item · 19

ADO.NET · 12

application · 7, 12, 22, 34, 49, 51, 59, 63, 64, 65, 66, 69, 70, 74

Authorization · 128

B

BuildUnityContainerMethod · 63

BusinessEntities · 13, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 47

BusinessServices · 34, 35, 36, 41, 47, 60

C

chapter · 8, 55, 57, 66, 70, 75

class library · 12, 13, 34

classes · 32, 35, 58, 62

code · 12, 17, 22, 28, 34, 35, 39, 40, 55, 59, 66, 67, 75

Conclusion · 54, 75

connection string · 15, 16

console · 21

context · 17, 20, 23, 28, 29, 30, 31

controller · 11, 44, 45, 46, 50, 65, 66, 67, 68, 70

CRUD · 8, 34, 36, 41, 47, 54, 339

D

Data Access Layer · 12

database · 8, 12, 14, 15, 16, 17, 20, 27, 28, 32, 39, 40, 52, 53, 54, 59, 70, 72, 73, 74

datamodel · 16, 17

DataModel · 12, 13, 20, 21, 22, 23, 28, 34, 35, 41, 60, 63, 75

Db context · 18

Delete · 24, 26, 39, 44, 47, 49, 53, 73

Dependencies · 76

Dependency Injection · 55, 56, 76

DependencyResolver · 61, 63

design · 22, 55, 57, 74, 75

E

edmx · 13, 14, 17, 19

Entity Framework · 7, 8, 12, 17, 18, 21

exception · 54

Exception Filters · 176

Exception Handling · 176

Extension manager · 35

Extension Manager · 17

G

Generic · 7, 12, 22, 23, 24, 25, 27, 28, 34, 35, 41, 46, 47

Generic Repository Pattern · 12

GenericRepository · 22, 23, 28, 29, 30

GET · 46, 48

H

HierarchicalLifetimeManager · 61, 62, 63

I

Install-Package · 21

Integration · 221, 278

©2016 Akhil Mittal (www.codet Teddy.com)

Inversion of Control · 54, 56, 75, 76

J

JSON · 8

L

Library Packet Manager · 21, 60

logging · 54

M

Managed Extensibility Framework · 76

MEF · 76

Microsoft SQL Server · 14

Moq · 221, 278

MVC · 10, 34, 99

MVC 4 · 10, 99

N

NLog · 176

O

objects · 12, 16, 26, 28, 32, 58, 59, 63, 66

OData · 311

ORM · 12

P

Packet Manager Console · 21, 60

product · 8, 29, 34, 36, 37, 38, 39, 40, 41, 42, 43, 44, 46, 47, 48, 49, 51, 52, 53, 58, 65, 71, 72, 73

ProductController · 46, 47, 48, 58, 65

ProductEntity · 33

ProductServices · 34, 35, 36, 41, 48, 59, 61, 62, 63, 65, 66

Project · 10, 34

properties · 20, 28, 29, 32, 59

R

repository · 22, 29, 39

Repository and Unit of Work · 22

REST · 8

Routing · 64, 99

S

security · 75

Setup · 8, 12, 32, 34, 44, 59, 65

T

table · 8, 34, 51

Test · 50, 70

token · 29

Tokens · 8

transactions · 20, 27, 28

U

Unit of Work · 7, *See*, *See*

UnitOfWork · 28, 31, 34, 35, 36, 41, 59, 60, 61, 62, 63, 65, 66, 75

Unity Container · 56, 57, 65, 76

Update · 25, 38, 43, 52, 72

URL · 51, 70, 99

V

Visual Studio · 9, 10, 12, 17, 18, 21, 59, 60, 339

Visual Studio 2010 · 9, 59

W

Web API · 8, 9, 11, 32, 35, 47, 50, 56, 57, 70, 128, 339

Web APIs · 7, 76, 99, 128, 176, 311

WebAPI · 10, 44, 45, 50, 54, 60, 61, 63, 66, 69, 221, 278

WebApiDataModel · 13

WebAPIdb · 15

WebApiDbEntities · 16, 23, 28, 29

Wikipedia · 8