RESTful Day #1: Enterprise level application architecture with Web API's using Entity Framework, Generic Repository pattern and Unit of Work. –by Akhil Mittal

Table of Contents

Introduction:

I have been practicing, reading a lot about RESTful services for past few days. To my surprise I could not find a complete series of practical implementations of ASP.NET Web API's on the web. My effort in this series will be to focus on how we can develop basic enterprise level application architecture with Web API's.

The complete series will be in a way that focuses on less theory and more practical scenarios to understand how RESTful services can be created using an ORM (Object-relational mapping), I choose Entity Framework here.
My first article in the series is to set up a basic architecture of REST service based application using Asp.net MVC.In this article I'll explain how to expose pure REST (Representational State Transfer) endpoints of the service that could be consumed by any client which wishes to use REST services. I'll explain the implementation of Entity Framework in conjunction with Repository Pattern and Unit of Work. I'll also focus on how to create a generic repository for all the entities that may come up as the application expands, in short it should be quite scalable and extensible.

Second article in the series talks about loosely coupled architecture. This article throws some light on how we can achieve a loosely coupled architecture in Asp.Net WebAPI's using UnityContainer and Bootstrapper. I'll try to implement Dependency Injection to achieve same.

My third article is about overcoming the flaws of UnityContainer. It explain how we can leverage MEF (Managed Extensibility Framework) to resolve the dependencies at runtime using IOC(Inversion of Control).

Unlike other web services, Asp.net WebAPI supports attribute based routing techniques .The fourth day of the series explains how we can get multiple endpoints of a WebAPI using Attribute Routing and also overcome of the traditional shortcomings of REST based services.

Security is always a concern in Enterprise level applications, Day #5 of the articles series explains how one can implement a custom token based authentication technique to make the API's more secure.

Sixth part of the article will explain the implementation of a centralized approach of logging and exception handling with the help of Action Filters and Exception filters. It teaches how to implement nLog in conjunction with Filters.

Developing an enterprise level infrastructure without Unit tests is worth not developing it. Day #7 teaches how to leverage nUnit to accomplish this task of Unit Testing in WebAPIs. I'll take live practical examples to do so in the seventh day of the article.

Day #8 of the series teaches the new emerging concept of OData. It teaches on how one can request a service with custom needs and requirements, on how one can implement OData support to WebAPIs.

Let's start with setting up the roadmap of learning,

Roadmap:

My road for the series is as follows,

- **RESTful Day #1: Enterprise level application architecture with Web API's using Entity Framework, Generic Repository pattern and Unit of Work.**
- RESTful Day #2: Inversion of control using dependency injection in Web API's using Unity Container and Bootstrapper.
- RESTful Day #3: Resolving dependency of dependencies with dependency injection in Web API's using Unity Container and Managed Extensibility Framework (MEF).
- RESTful Day #4: Custom URL re-writing with the help of Attribute routing in MVC 4 Web API's.
- RESTful Day #5: Token based custom authorization in Web API's using Action Filters.
- RESTful Day #6: Request logging and Exception handing/logging in Web API's using Action Filters, Exception Filters and nLog.
- RESTful Day #7: Unit testing Asp.Net Web API's controllers using nUnit.
- RESTful Day #8: Extending OData support in Asp.Net Web API's.

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

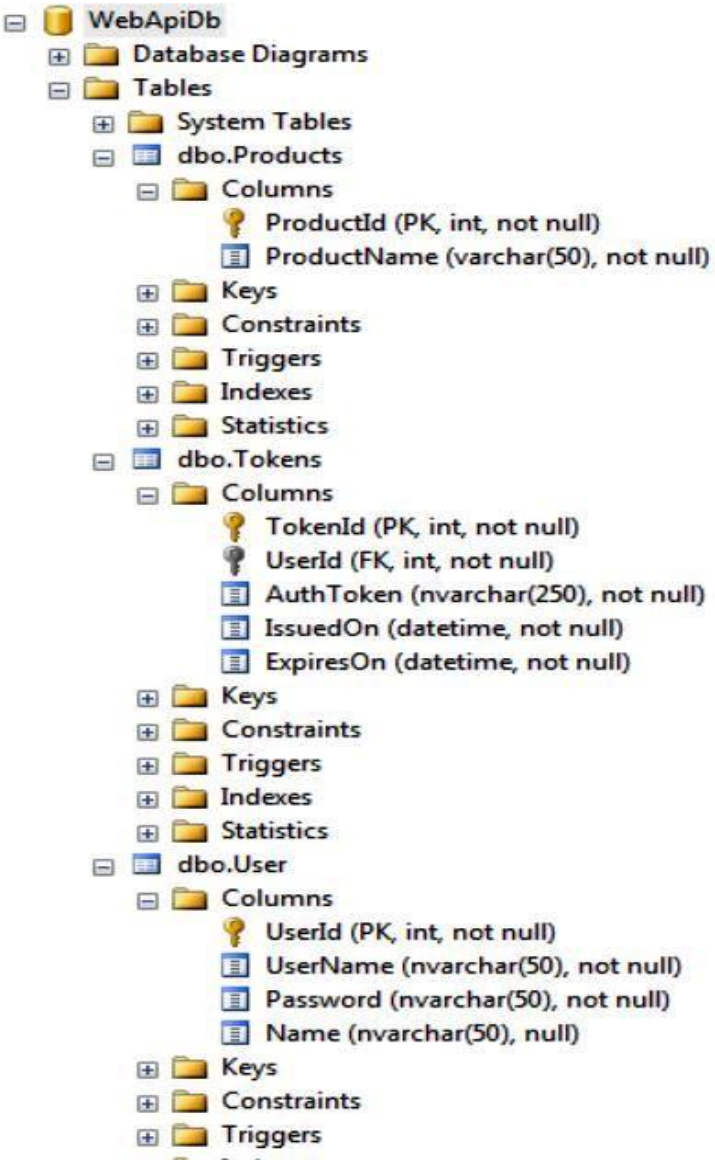
REST:

Here is an extract from Wikipedia,

*"Unlike SOAP-based web services, there is no "official" standard for **RESTful** web APIs. This is because REST is an architectural style, while 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:

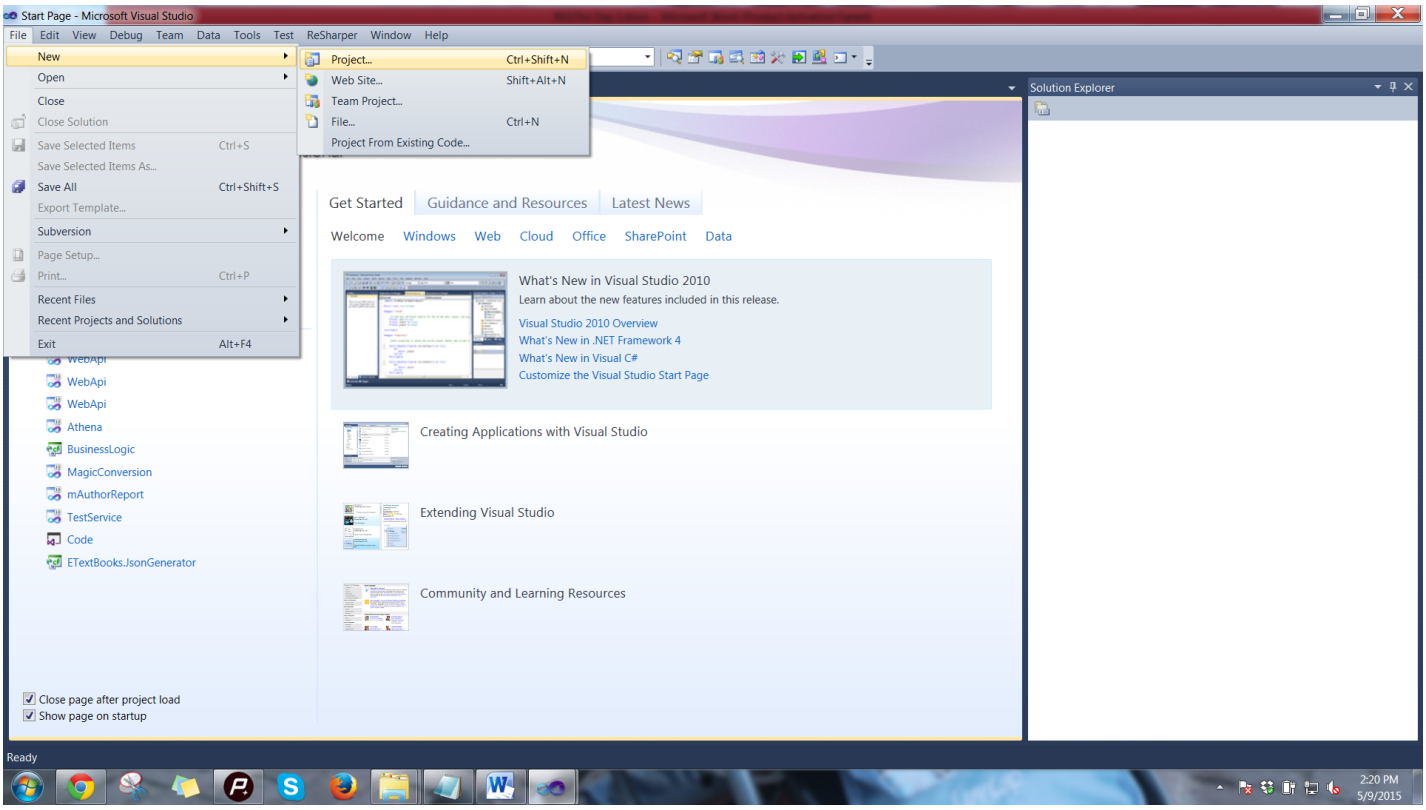
I am using Sql Server 2008 as a database server. I have provided the sql scripts to create the database in Sql Server, you can use the same to create one. I have given WebApiDb as my database name.My database contains three tables for now, Products, Tokens, User. In this tutorial we'll only be dealing with product table to perform CURD operations using Web API and Entity framework.We'll use Tokens and User in my upcoming article. For those who fail to create database through scripts, here is the structure you can follow,



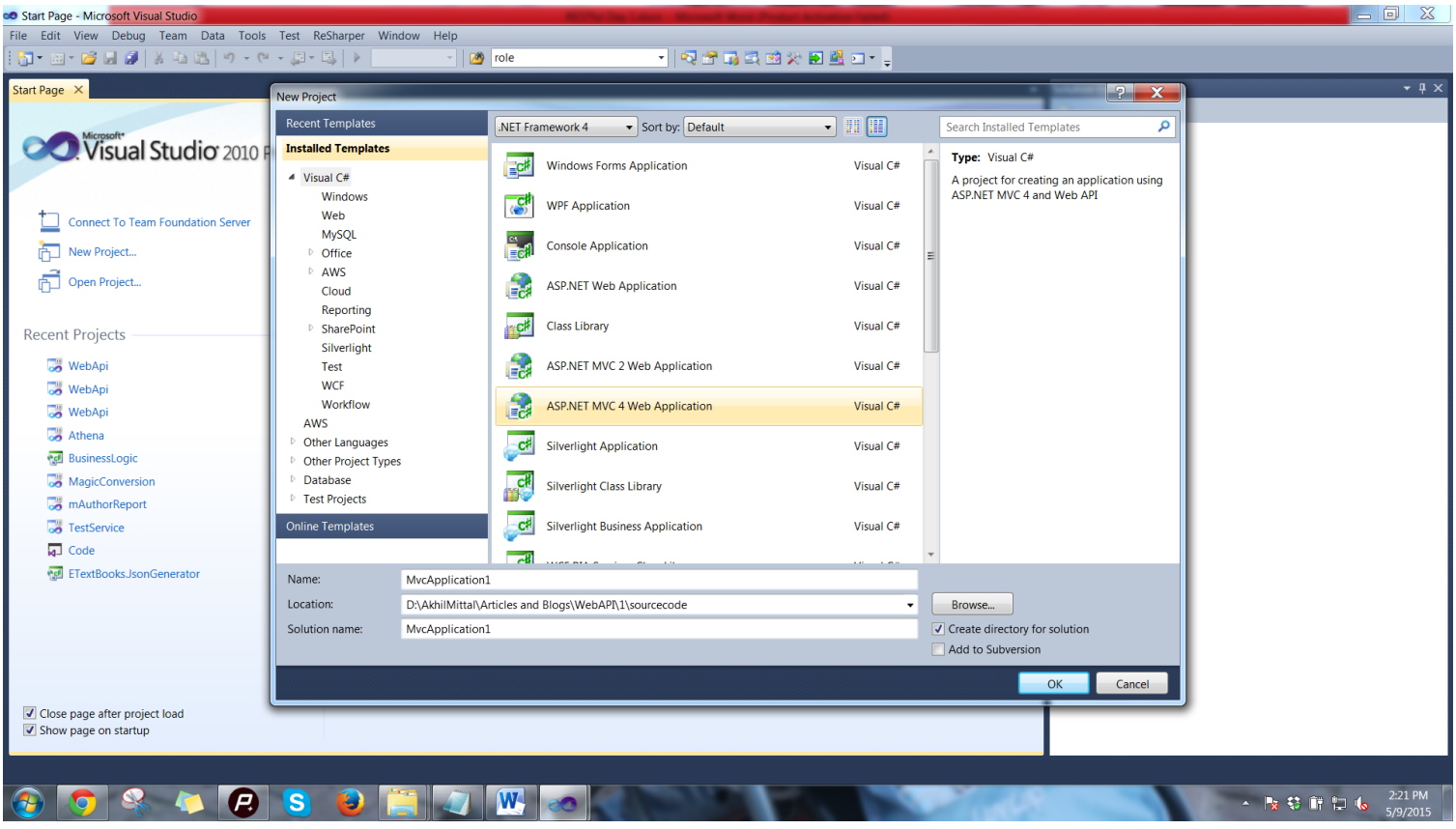
Web API project:

Open your Visual studio , I am using VS 2010, You can use VS version 2010 or above.

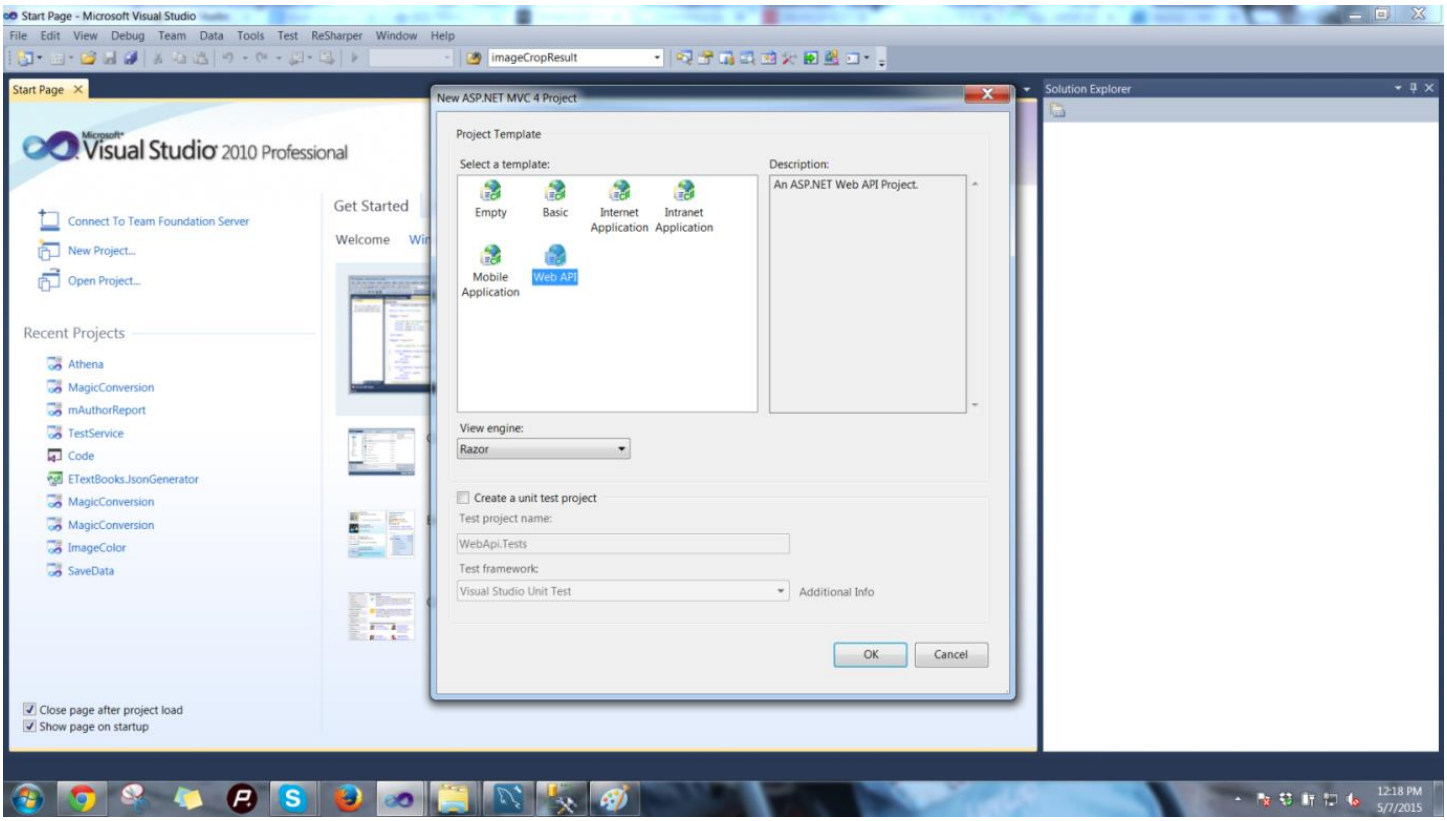
Step 1: Create a new Project in your visual studio,



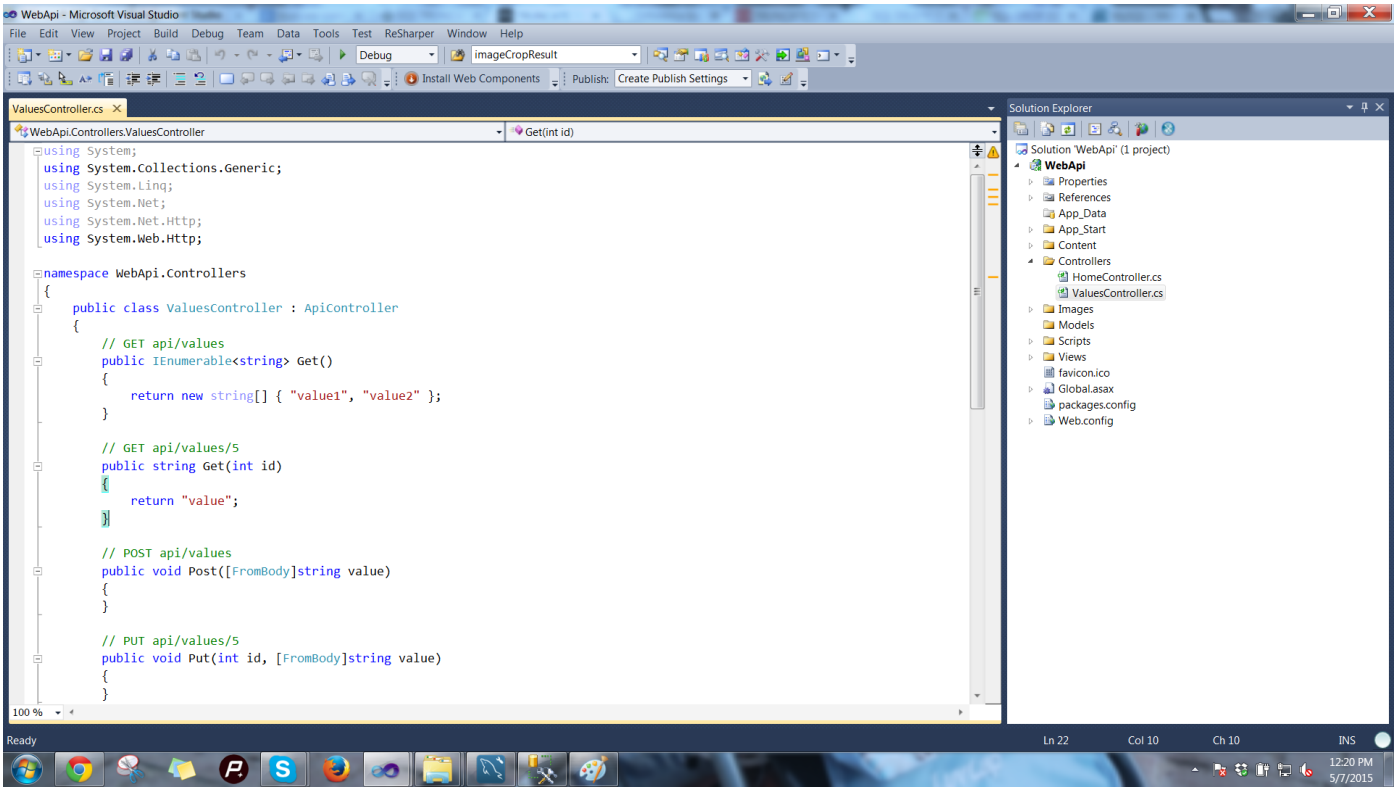
Step 2: There after choose to create Asp.net MVC 4 Web application, and give it a name of your choice, I gave it **WebAPI**.



Step 3: Out of different type of project templates shown to you, choose Web API project,



Once done, you'll get a project structure like shown below, with a default Home and Values controller. You can choose to delete this ValuesController , as we'll be using our own controller to learn.



Setup Data Access Layer:

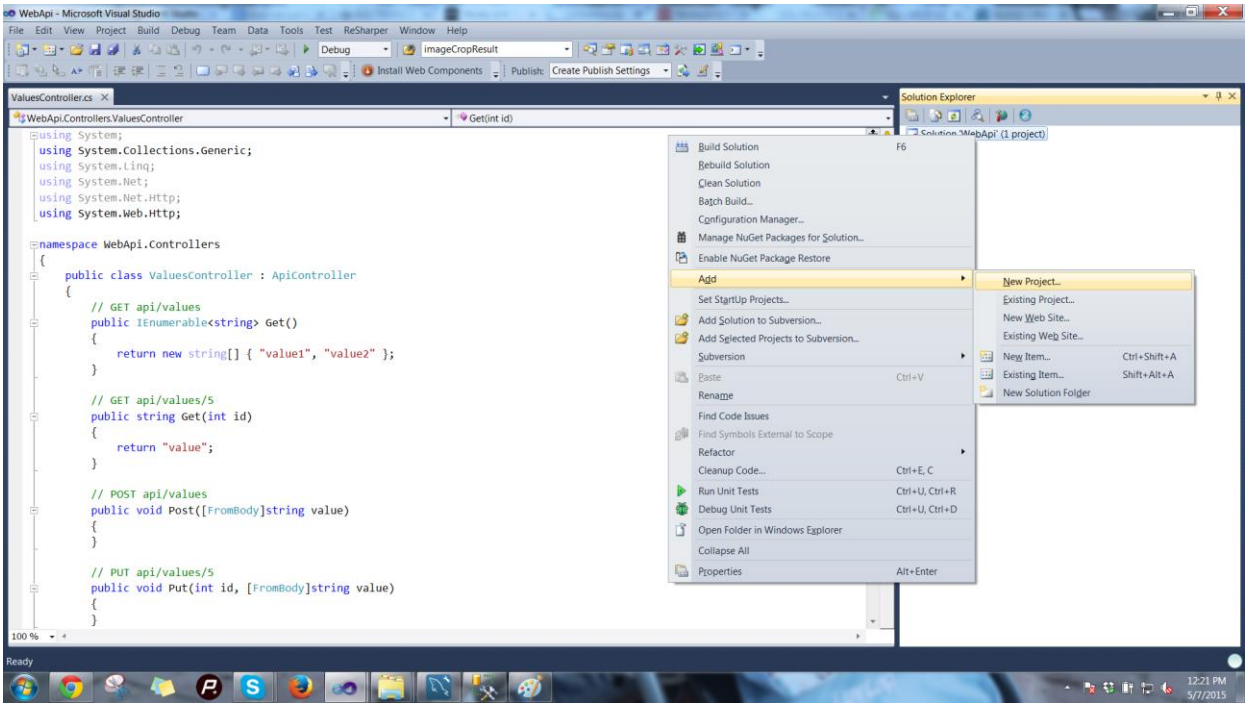
Let's setup or data access layer first. We'll be using Entity Framework 5.0 to talk to database. We'll use Generic Repository Pattern and Unit of work pattern to standardize our layer.

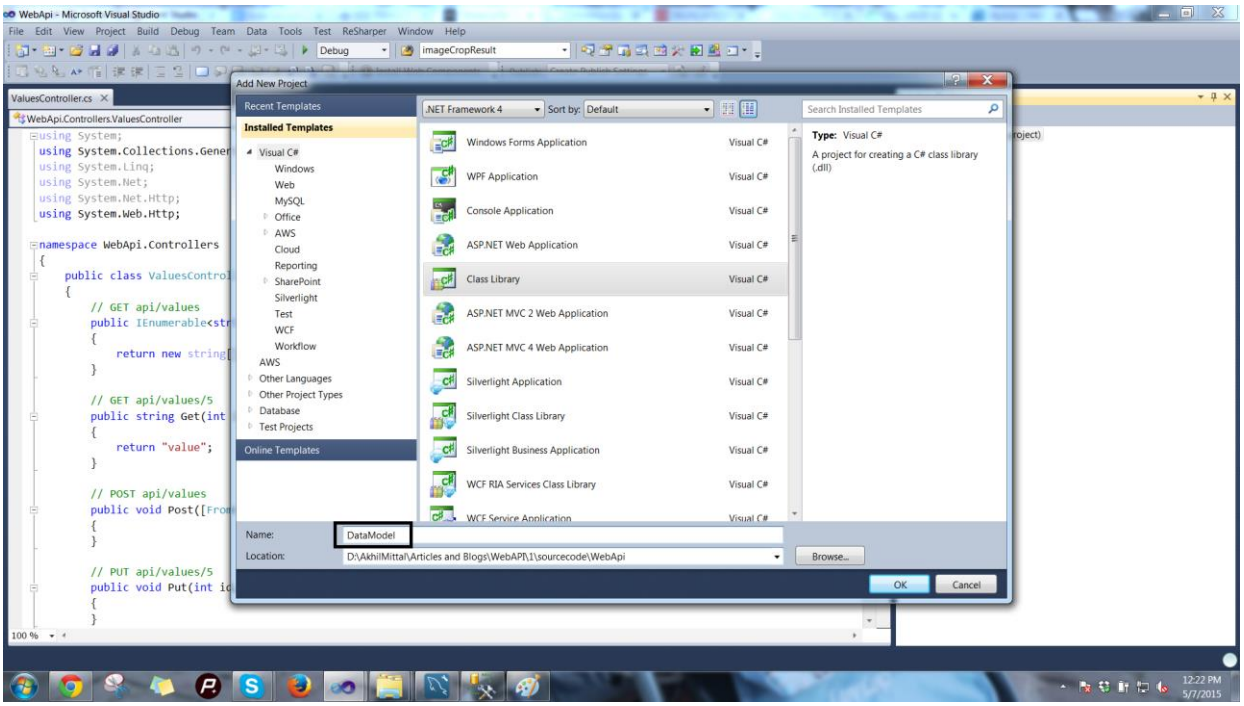
Let's have a look at the standard definition of Entity Framework given by Microsoft:

"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."

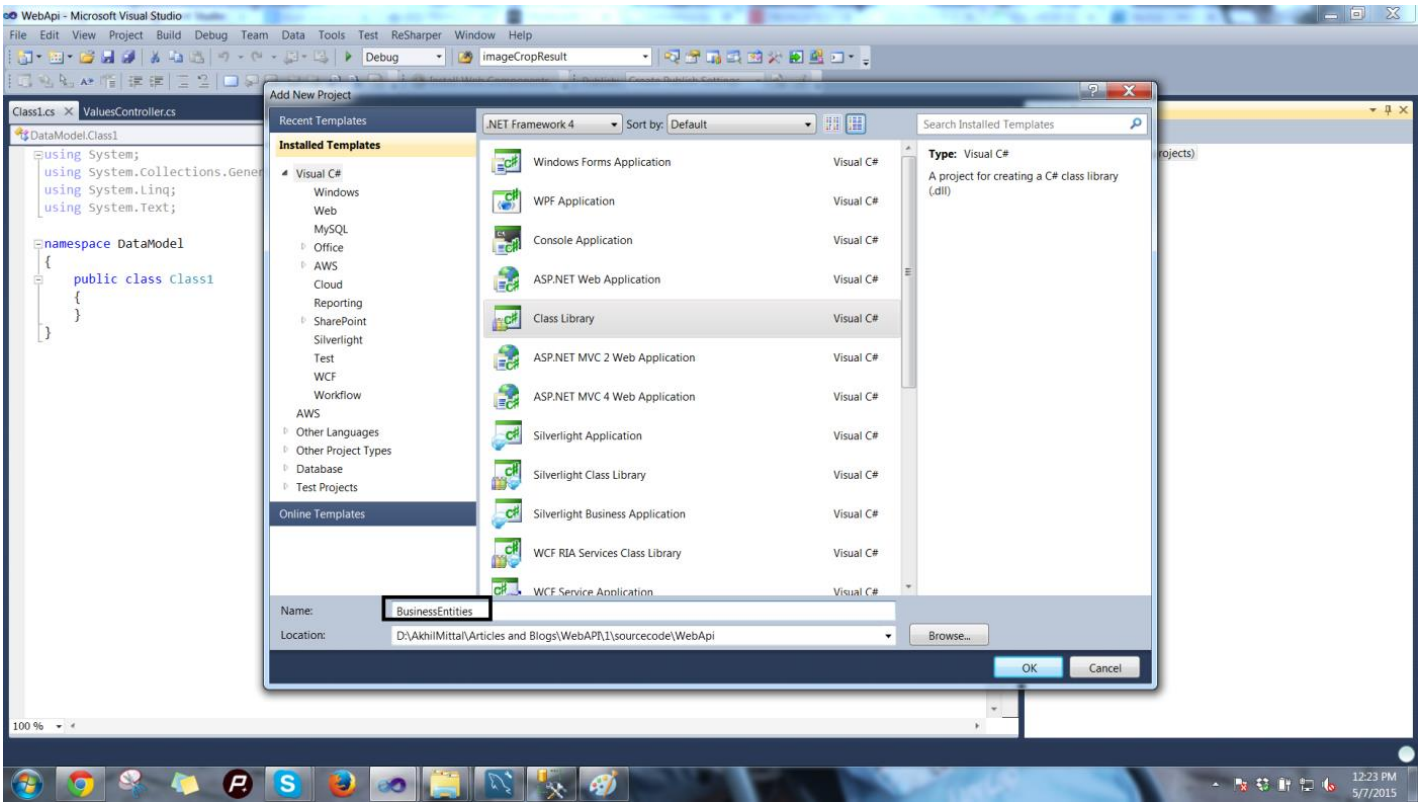
In simple language, Entity framework is an Object/Relational Mapping (ORM) framework. It is an enhancement to ADO.NET, an upper layer to ADO.NET that gives developers an automated mechanism for accessing and storing the data in the database.

Step 1 : Create a new class library in your visual studio, and name it DataModel as shown below,



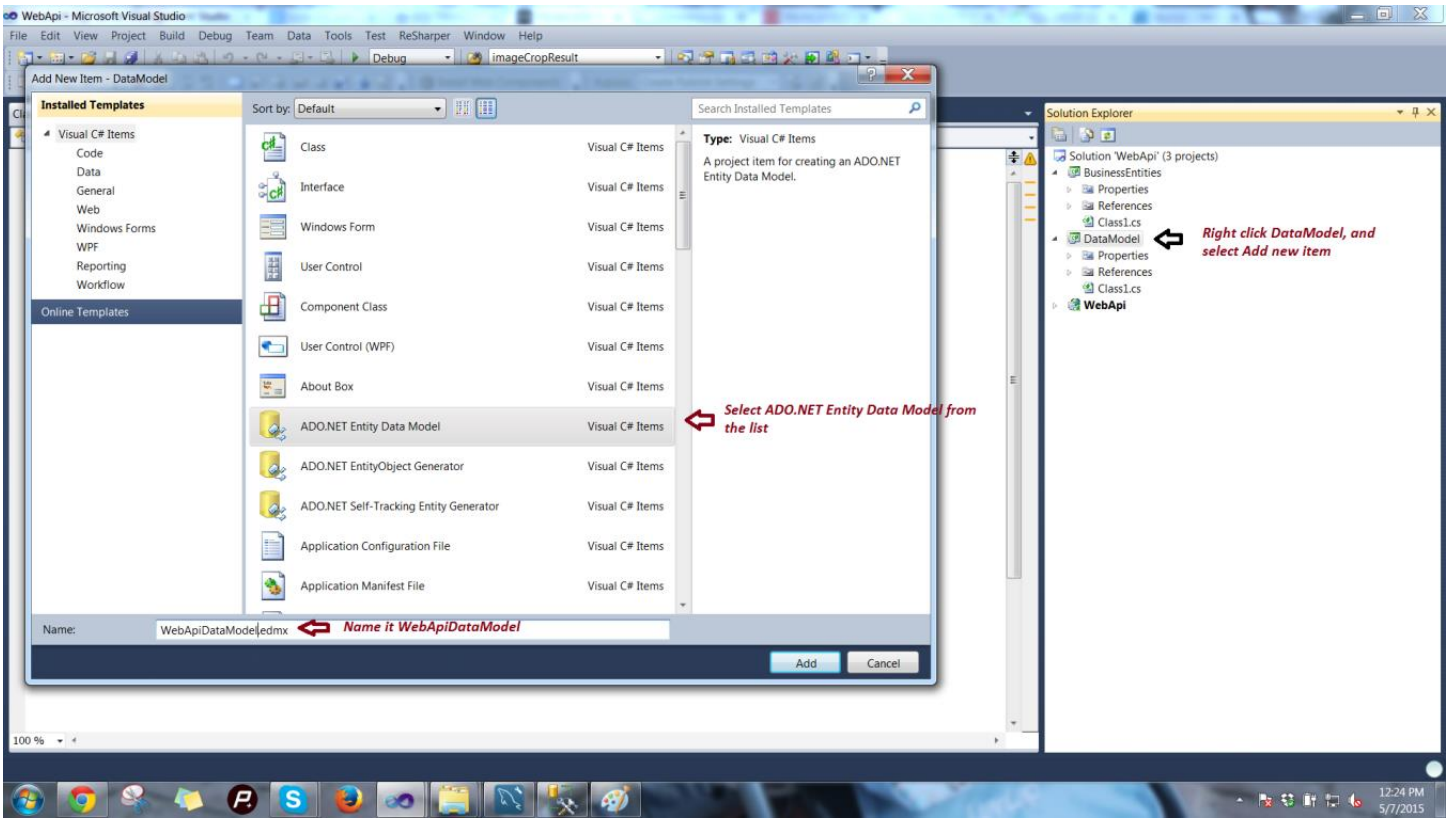


Step2: In the same way, create one more project i.e. again a class library and call it BusinessEntities,

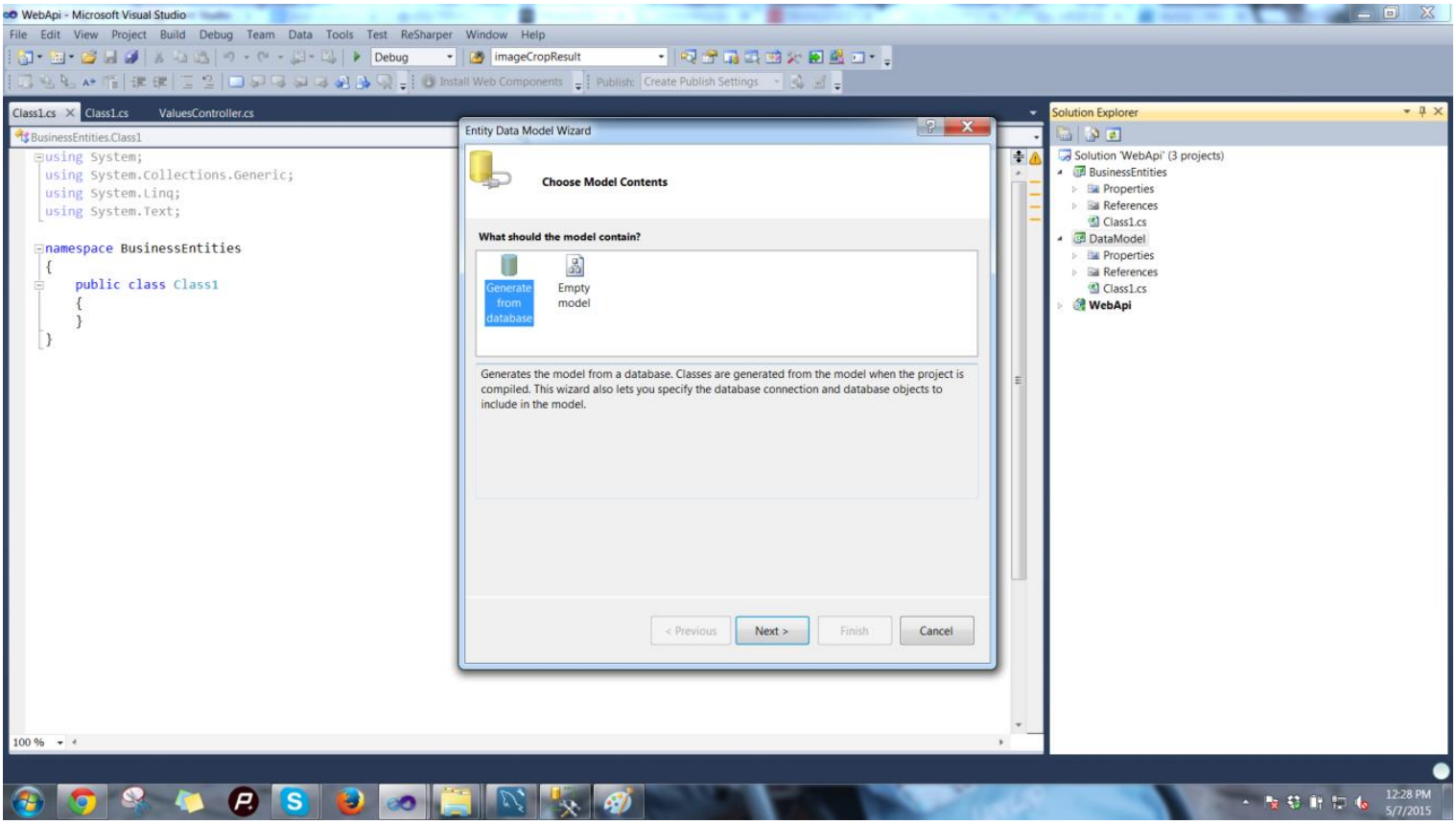


I'll explain the use of this class library soon.

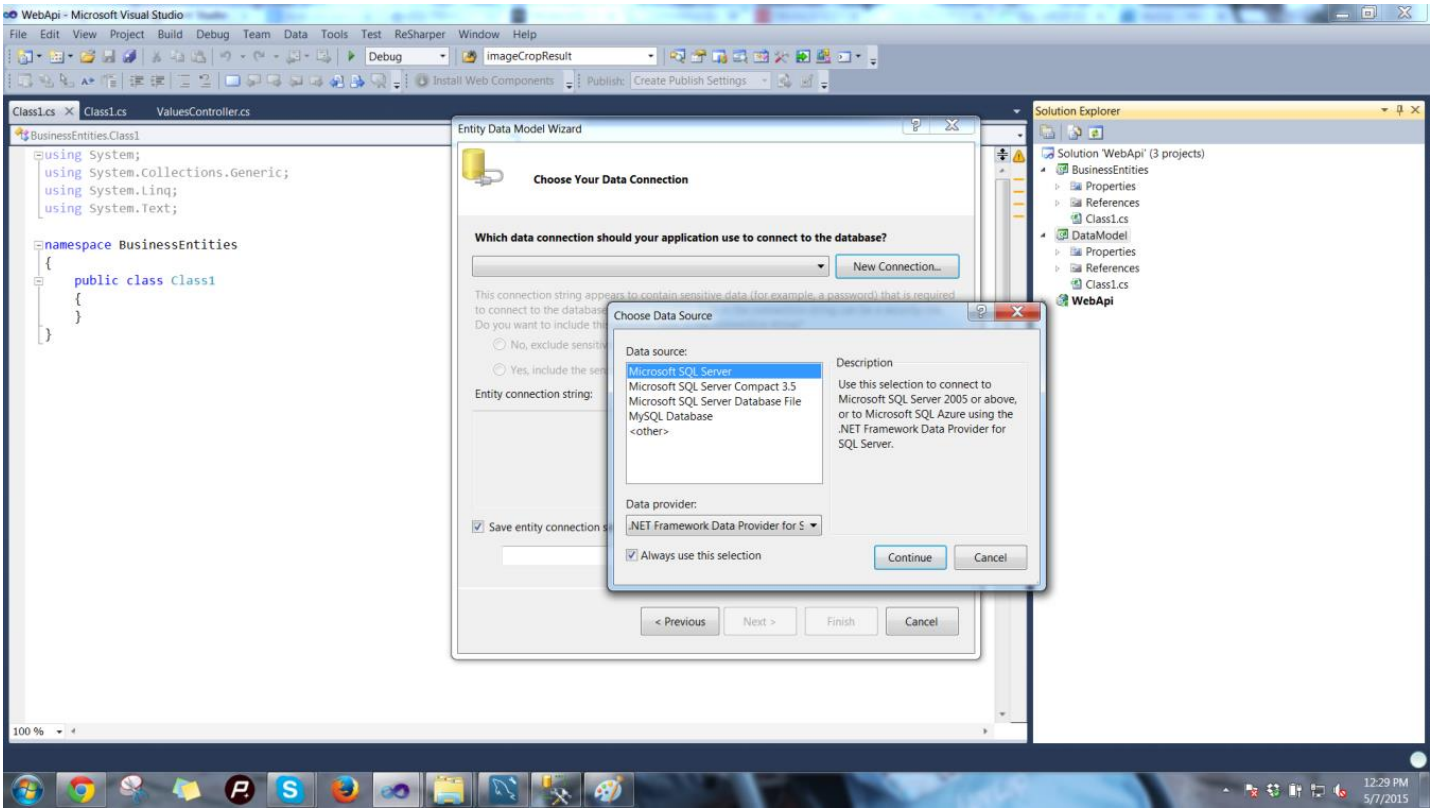
Step 3: Move on to your DataModel project, right click on it and add a new item, in the list shown, choose ADO.net Data Model, and name it WebApiDataModel.edmx.



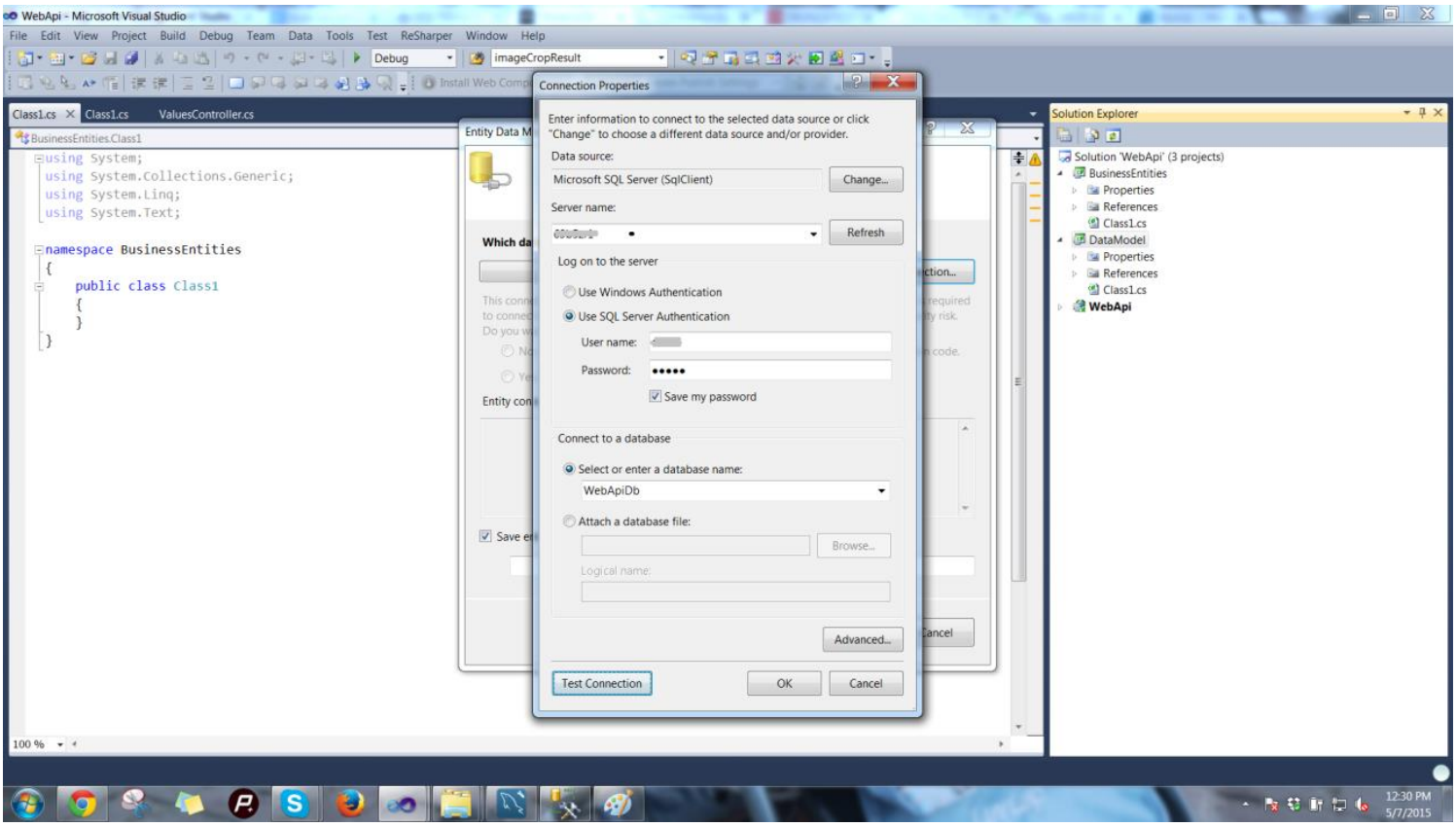
The file .edmx will contain the databse information of our database that we created earlier, let's set up this. You'll be presented a wizard like follows,



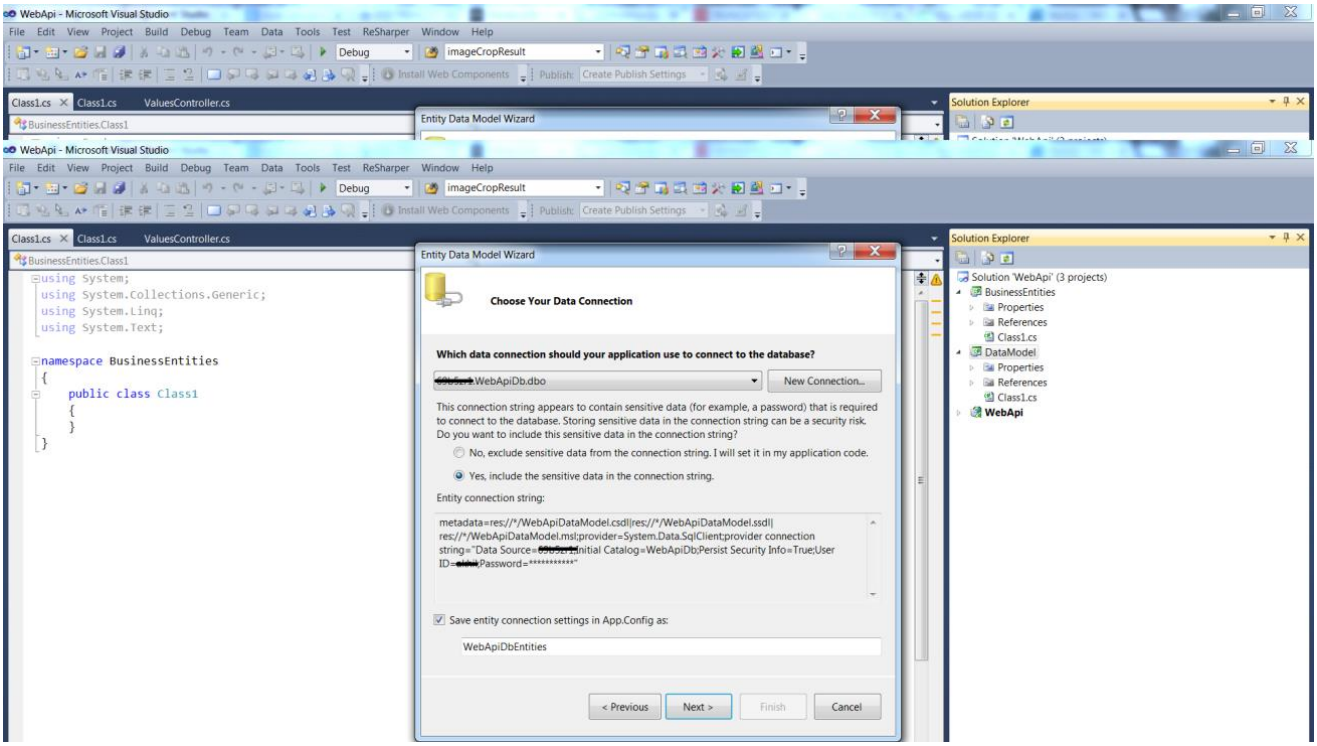
Choose, generate from database. Choose Microsoft SQl Server like shown in the following image,



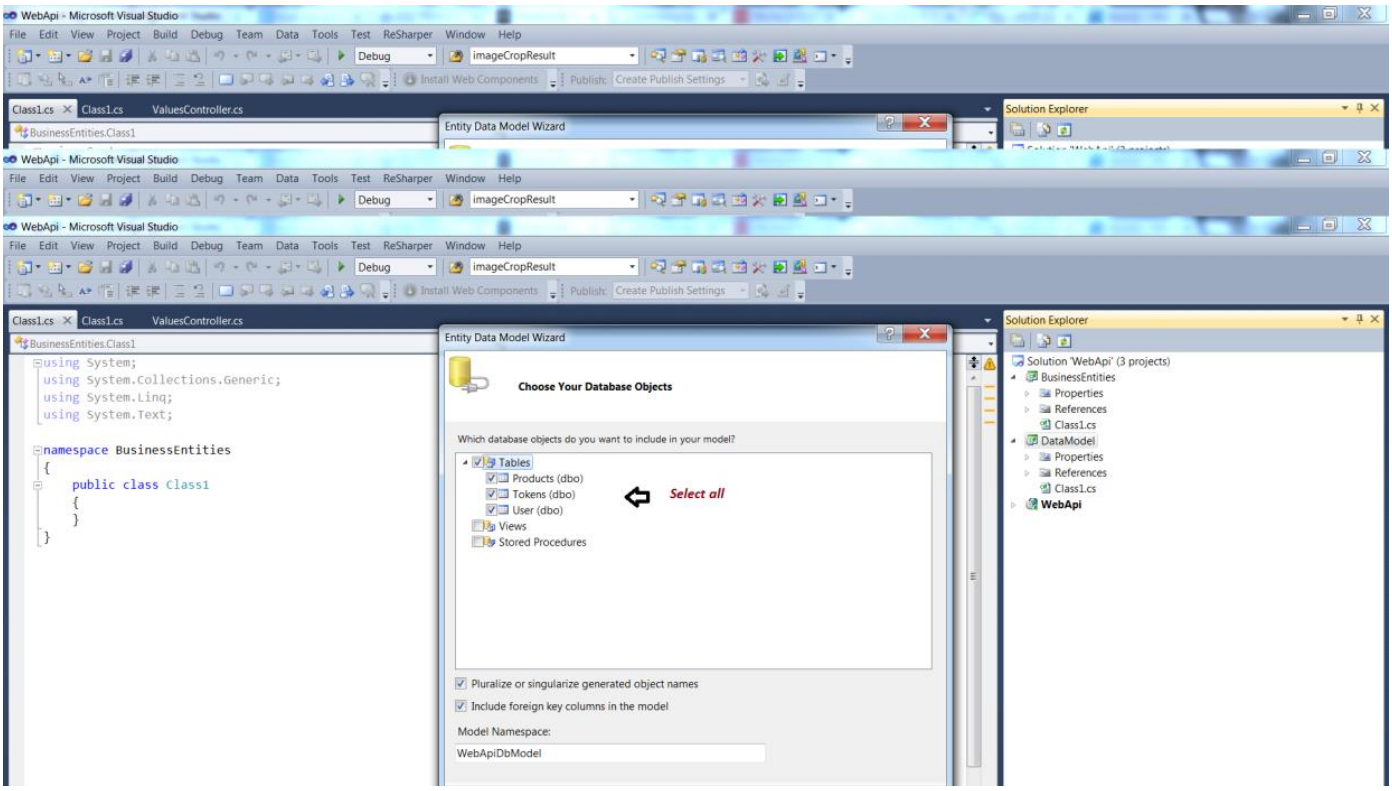
Click continue, then provide the credentials of your database, i.e. WebAPIdb, and connect it,



You'll get a screen, showing the connection string of the database we chose,

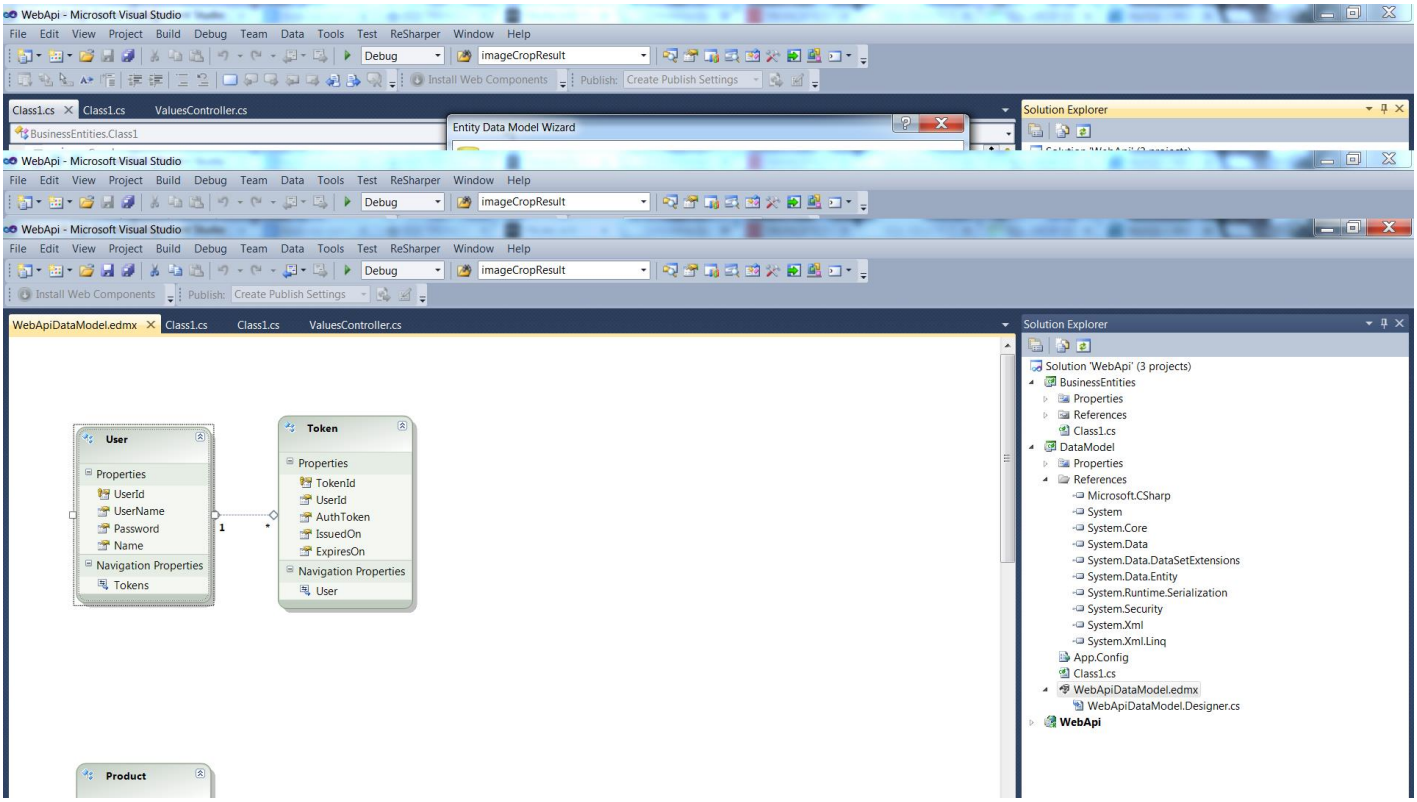


Provide the name of the connection string as WebApiDbEntities and click Next.



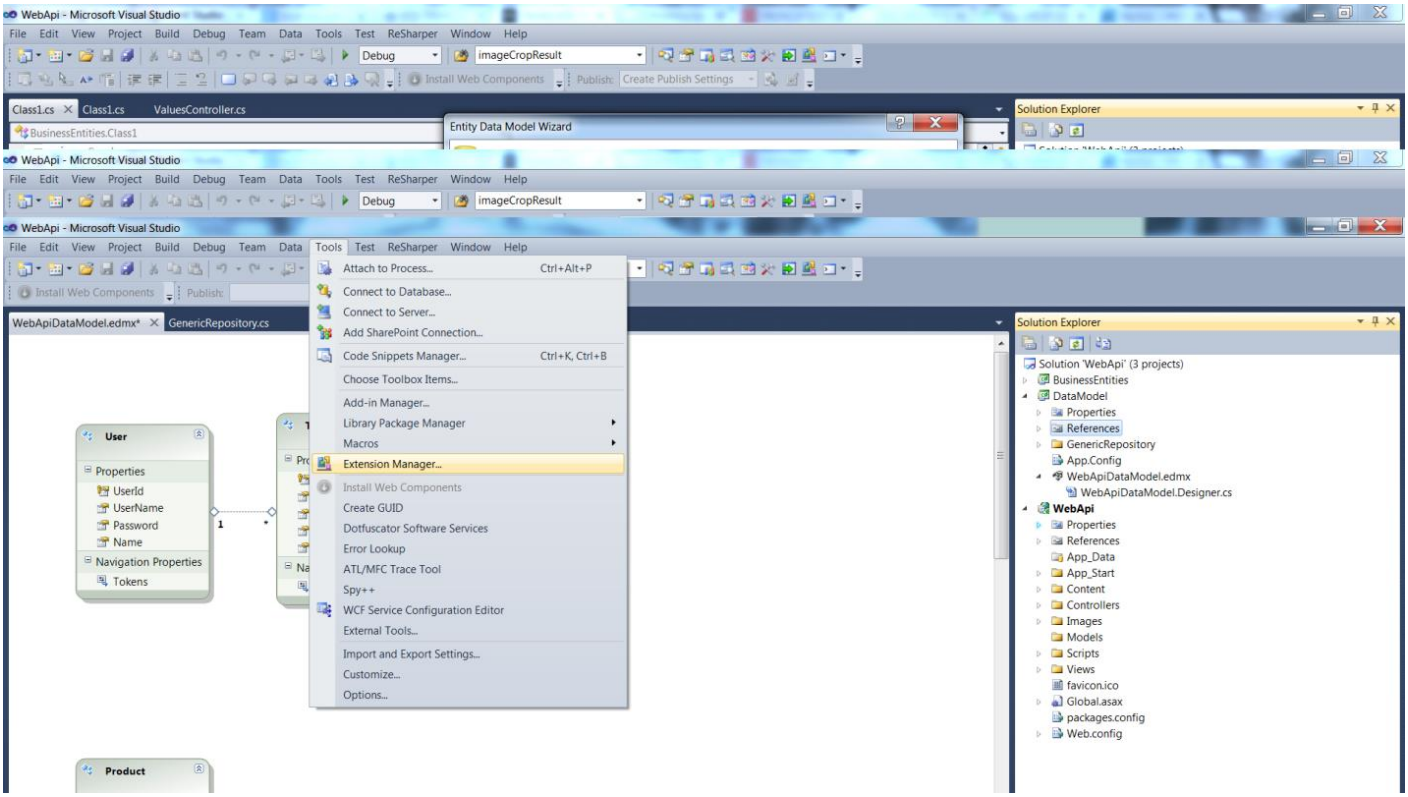
Choose all the database objects, check all the check boxes, and provide a name for the model. I gave it a name WebApiDbModel.

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

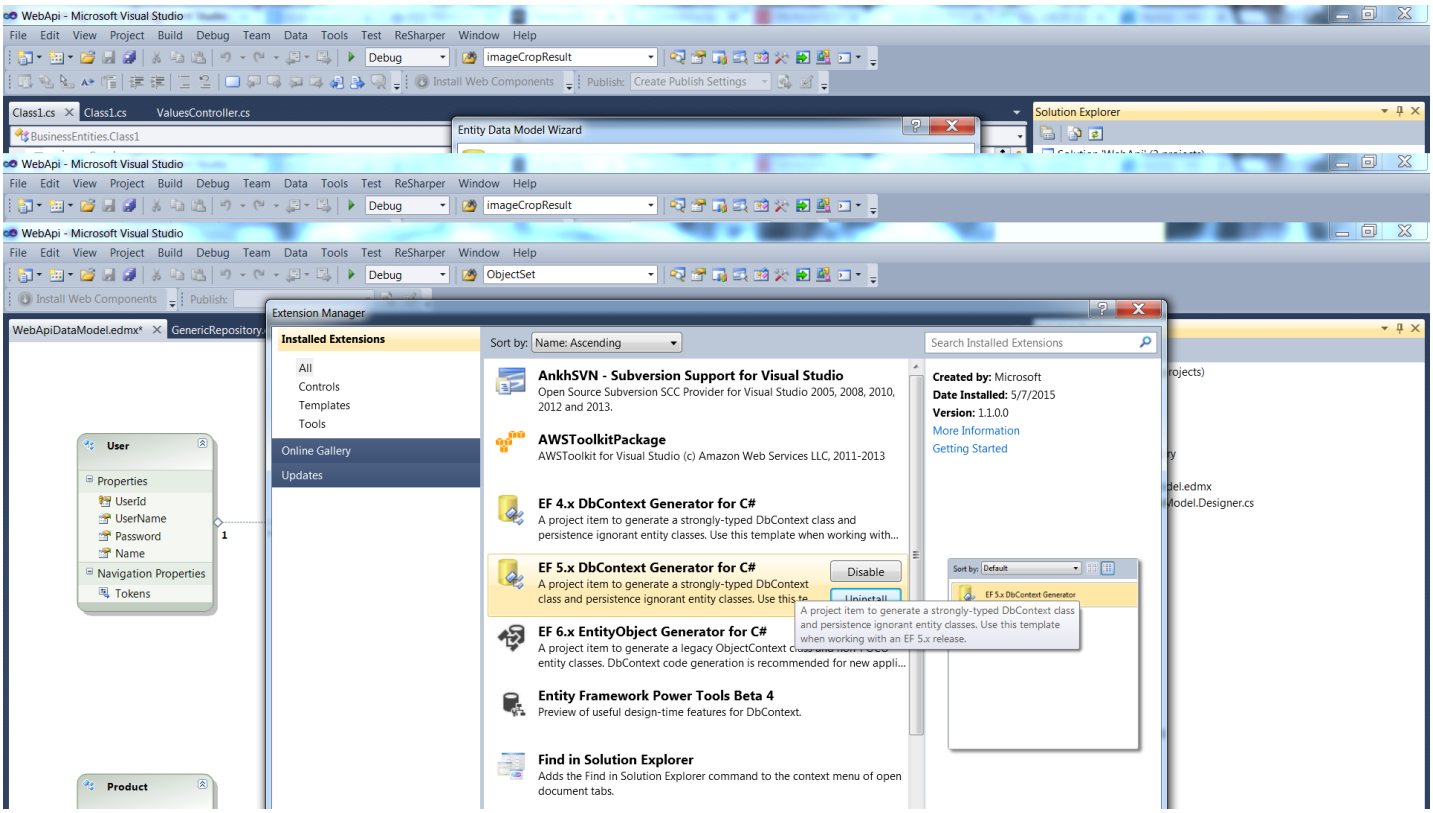


We've got our schema in-place using Entity framework. But few work is still remaining. We need our data context class and entities through which we'll communicate with database.
So, moving on to next step.

Step 3 : Click on tools in Visual studio and open Extension manager. We need to get db context generator for our datamodel.We can also do it using default code generation item by right clicking in the edmx view and add code generation item, but that will generate object context class and that is heavier than db context. I want light weighted db context class to be created, so we'll use extension manager to add a package and then create a db context class.

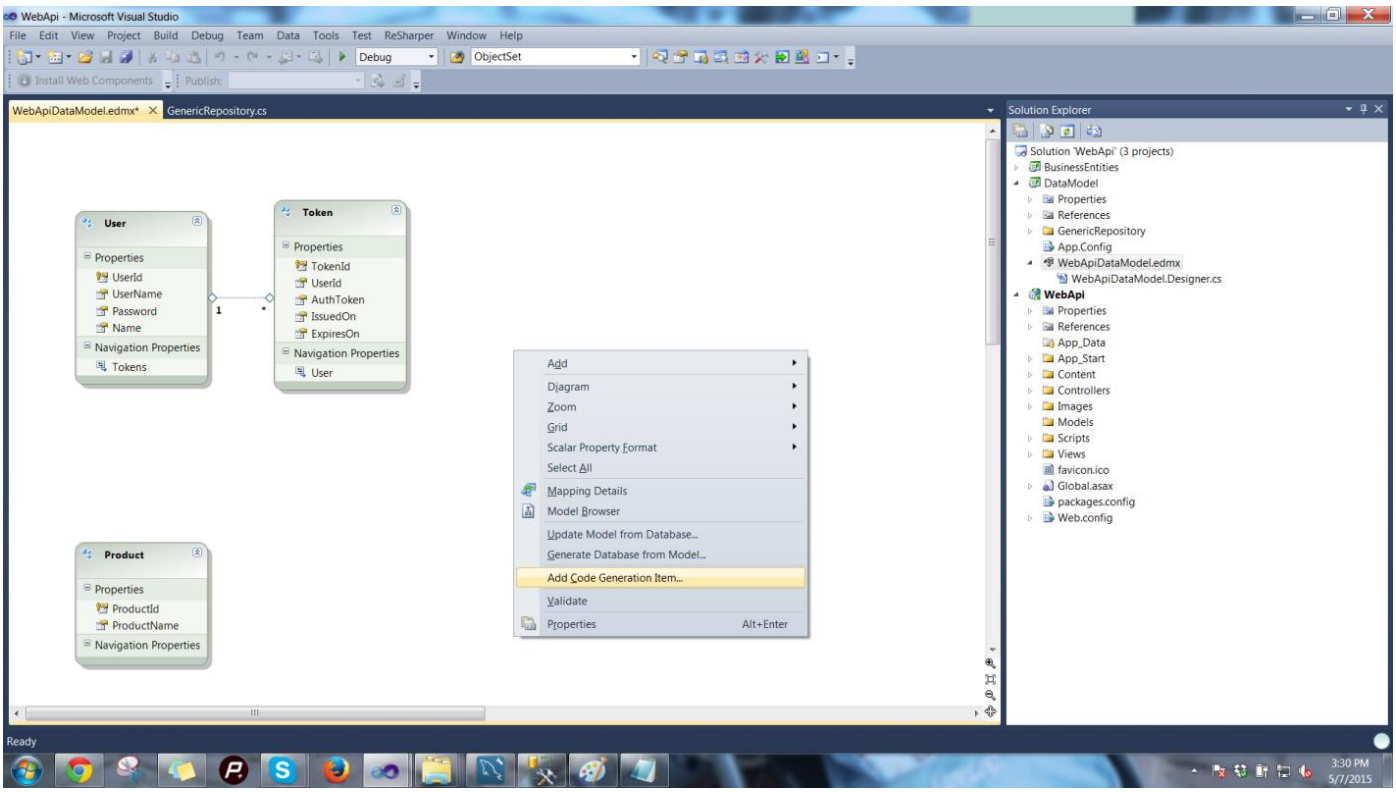


Search for Entity Framework Db context generator in online galary and select the one for EF 5.x like below,

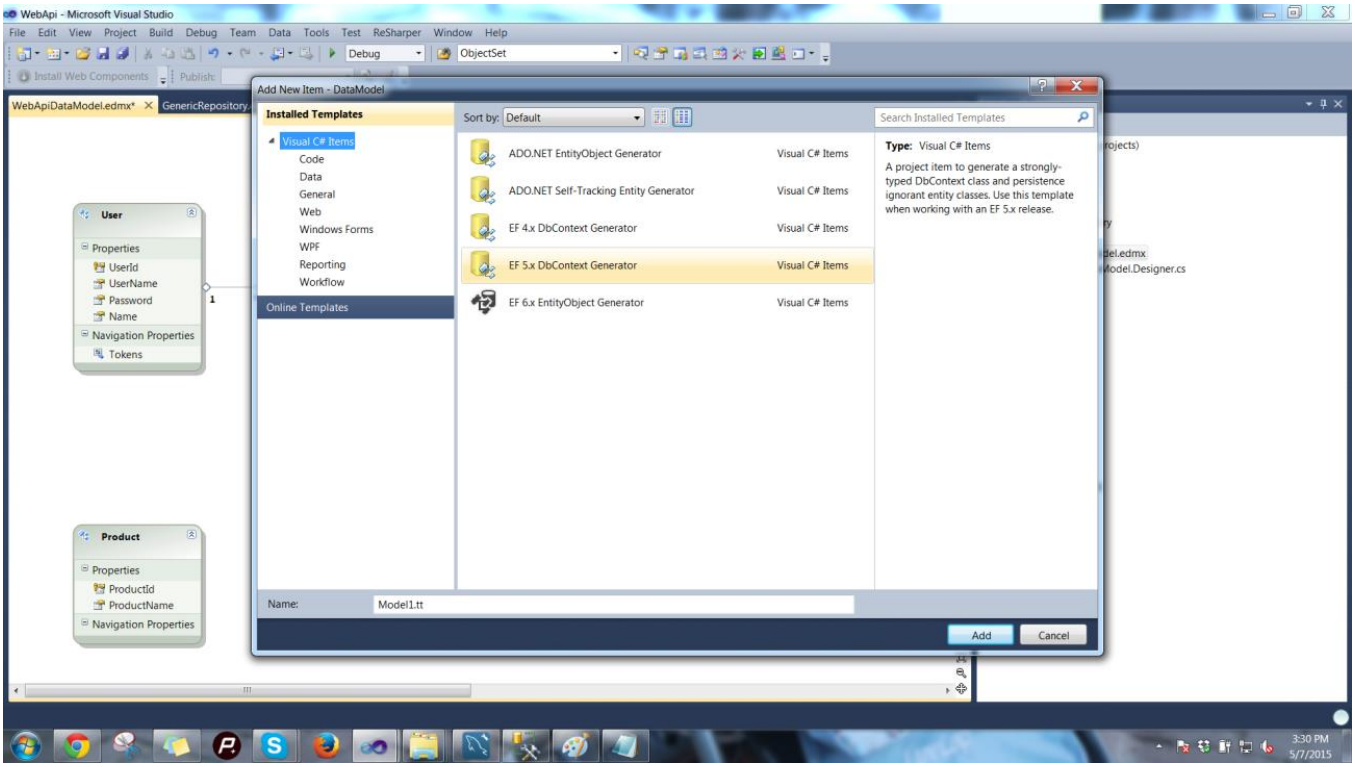


I guess you need to restart Visual studio to get that into your templates.

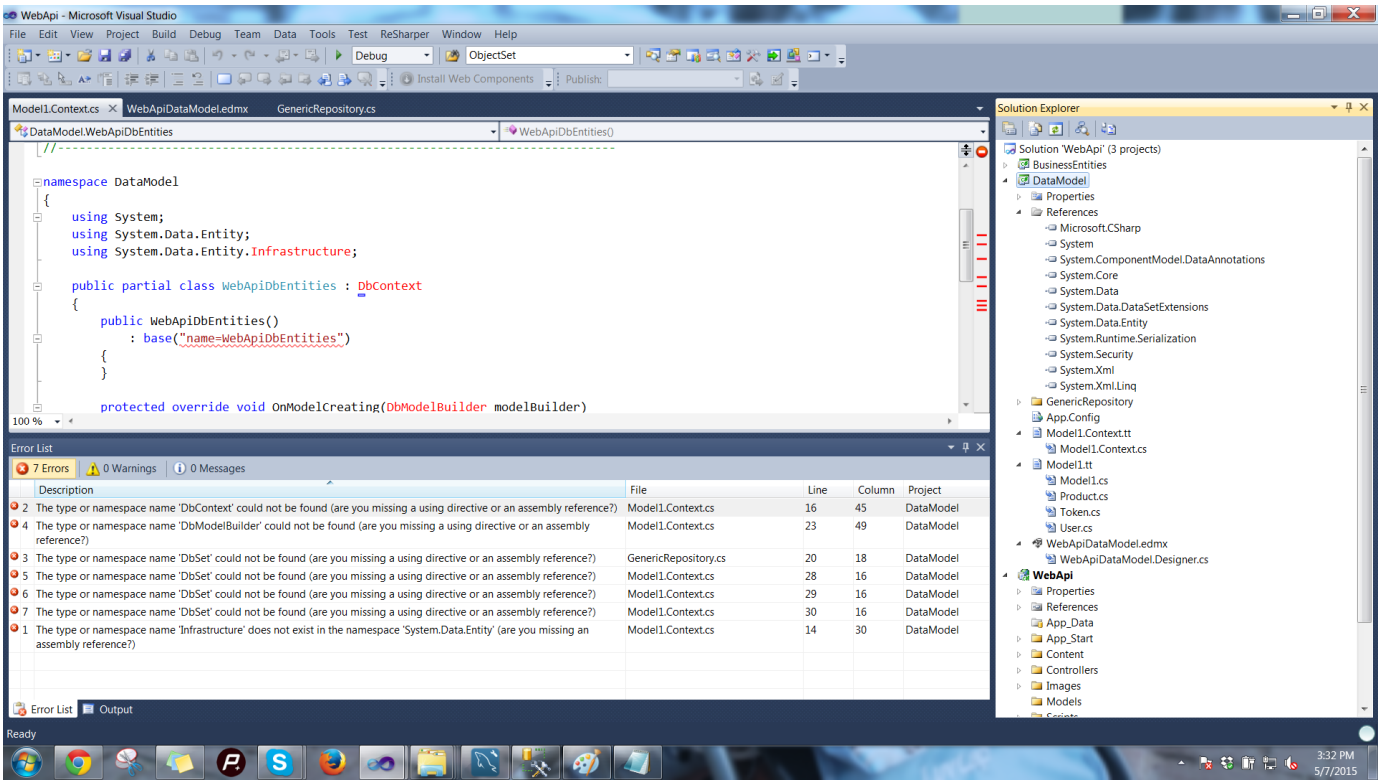
Step 4 : Now right click in the .edmx file schema designer and choose “Add Code Generation Item..”.



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

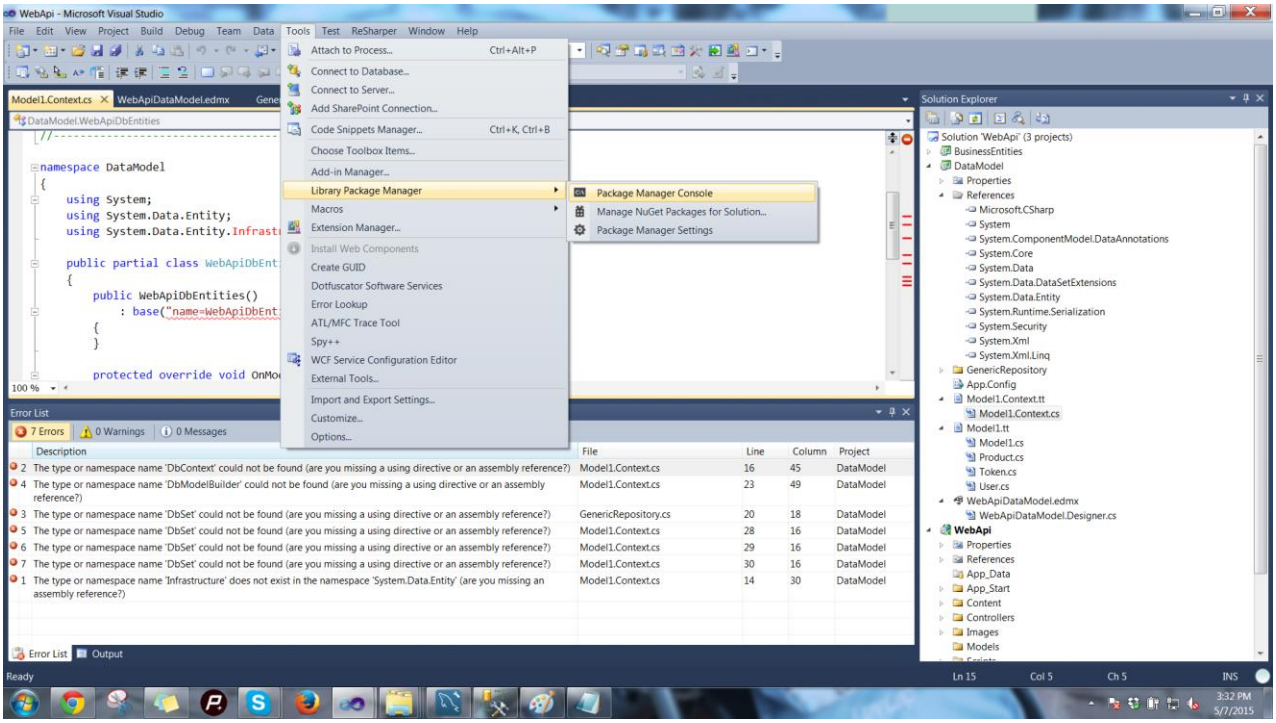


After adding this we'll get the db context class and its properties, this class is responsible for all database transactions that we need to perform, so our structure looks like as shown below,

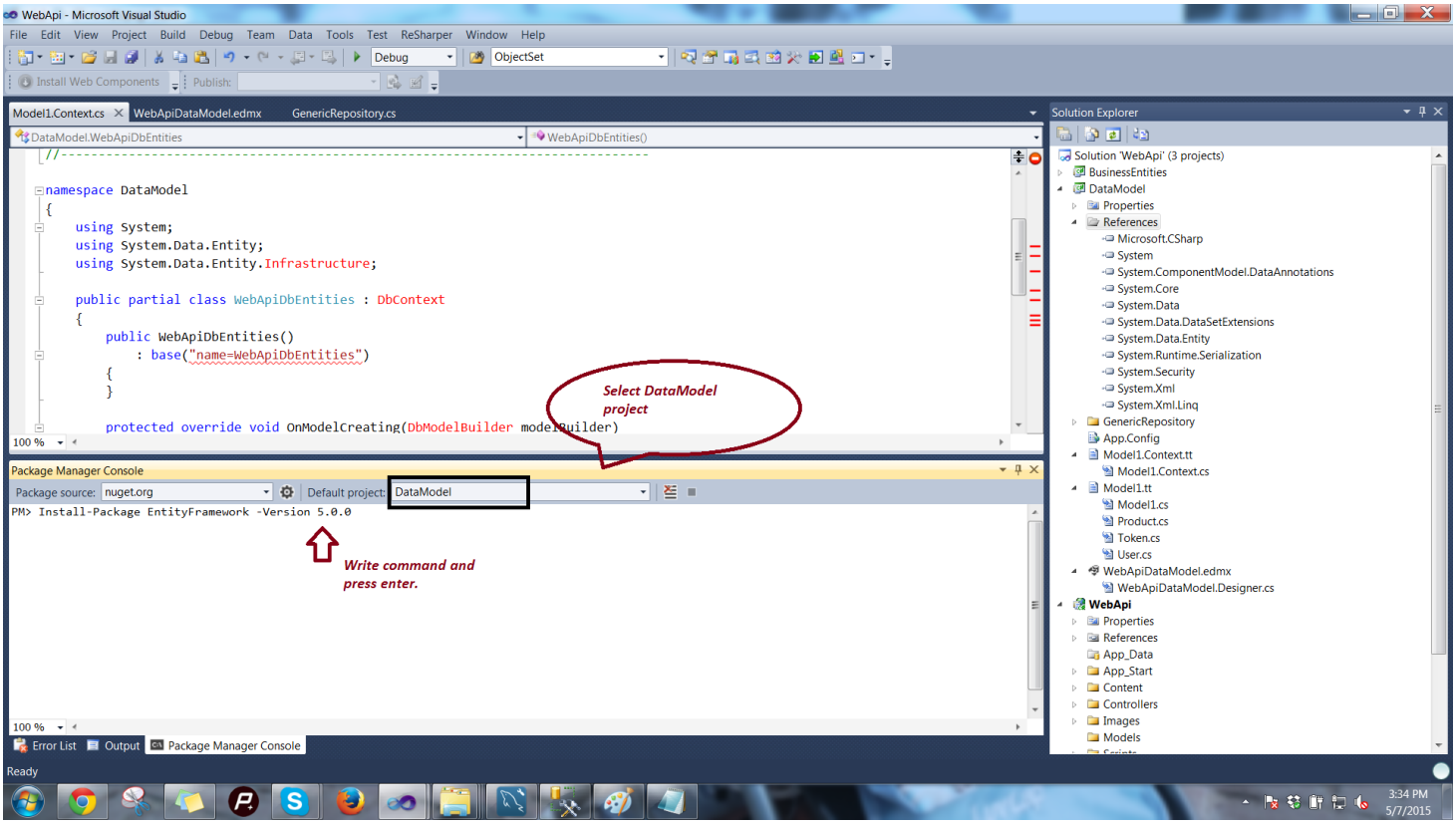


Wow, we ended up in errors. But we got our db context class and our entity models, You can see them in our DataModel project. Errors ? Nothing to worry about , it's just we did not reference entity framework in our project.We'll do it right away.

Step 6 : Go to Tools -> Library Packet Manager->Packet manager Console.
You'll get the console in left bottom of Visual studio.



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



Press enter. And all the errors get resolved.

Generic Repository and Unit of Work:

You can read about repository pattern and creating a repository in detail from my article http://www.codeproject.com/Articles/631668/Learning-MVC-Part-Repository-Pattern-in-MVC-App.

Just to list down the benefits of 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. For creating Generic Repository you can follow : http://www.codeproject.com/Articles/640294/Learning-MVC-Part-Generic-Repository-Pattern-in

Step 1: Add a folder named GenericRepository in DataModel project and to that folder add a class named Generic Repository. Add following code to that class, that servers as a template based generic code for all the entities that will interact with databse,

```
#region Using Namespaces...

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

#endregion

namespace DataModel.GenericRepository
{
    /// <summary>
    /// Generic Repository class for Entity Operations
    /// </summary>
    /// <typeparam name="TEntity"></typeparam>
    public class GenericRepository<TEntity> where TEntity : class
    {
        #region Private member variables...
        internal WebApiDbEntities Context;
        internal DbSet<TEntity> DbSet;
        #endregion

        #region Public Constructor...
        /// <summary>
        /// Public Constructor,initializes privately declared local variables.
        /// </summary>
        /// <param name="context"></param>
        public GenericRepository(WebApiDbEntities context)
        {
            this.Context = context;
            this.DbSet = context.Set<TEntity>();
        }
        #endregion

        #region Public member methods...

        /// <summary>
        /// generic Get method for Entities
        /// </summary>
        /// <returns></returns>
        public virtual IEnumerable<TEntity> Get()
        {
            IQueryable<TEntity> query = DbSet;
            return query.ToList();
        }

        /// <summary>
        /// Generic get method on the basis of id for Entities.
        /// </summary>
        /// <param name="id"></param>
```

```
/// <returns></returns>
public virtual TEntity GetByID(object id)
{
    return DbSet.Find(id);
}

/// <summary>
/// generic Insert method for the entities
/// </summary>
/// <param name="entity"></param>
public virtual void Insert(TEntity entity)
{
    DbSet.Add(entity);
}

/// <summary>
/// Generic Delete method for the entities
/// </summary>
/// <param name="id"></param>
public virtual void Delete(object id)
{
    TEntity entityToDelete = DbSet.Find(id);
    Delete(entityToDelete);
}

/// <summary>
/// Generic Delete method for the entities
/// </summary>
/// <param name="entityToDelete"></param>
public virtual void Delete(TEntity entityToDelete)
{
    if (Context.Entry(entityToDelete).State == EntityState.Detached)
    {
        DbSet.Attach(entityToDelete);
    }
    DbSet.Remove(entityToDelete);
}

/// <summary>
/// Generic update method for the entities
/// </summary>
/// <param name="entityToUpdate"></param>
public virtual void Update(TEntity entityToUpdate)
{
    DbSet.Attach(entityToUpdate);
    Context.Entry(entityToUpdate).State = EntityState.Modified;
}

/// <summary>
/// generic method to get many record on the basis of a condition.
/// </summary>
/// <param name="where"></param>
/// <returns></returns>
public virtual IEnumerable<TEntity> GetMany(Func<TEntity, bool> where)
{
    return DbSet.Where(where).ToList();
```

```
        }

        /// <summary>
        /// generic method to get many record on the basis of a condition but query able.
        /// </summary>
        /// <param name="where"></param>
        /// <returns></returns>
        public virtual IQueryable<TEntity> GetManyQueryable(Func<TEntity, bool> where)
        {
            return DbSet.Where(where).AsQueryable();
        }

        /// <summary>
        /// generic get method , fetches data for the entities on the basis of condition.
        /// </summary>
        /// <param name="where"></param>
        /// <returns></returns>
        public TEntity Get(Func<TEntity, Boolean> where)
        {
            return DbSet.Where(where).FirstOrDefault<TEntity>();
        }

        /// <summary>
        /// generic delete method , deletes data for the entities on the basis of condition.
        /// </summary>
        /// <param name="where"></param>
        /// <returns></returns>
        public void Delete(Func<TEntity, Boolean> where)
        {
            IQueryable<TEntity> objects = DbSet.Where<TEntity>(where).AsQueryable();
            foreach (TEntity obj in objects)
                DbSet.Remove(obj);
        }

        /// <summary>
        /// generic method to fetch all the records from db
        /// </summary>
        /// <returns></returns>
        public virtual IEnumerable<TEntity> GetAll()
        {
            return DbSet.ToList();
        }

        /// <summary>
        /// Inclue multiple
        /// </summary>
        /// <param name="predicate"></param>
        /// <param name="include"></param>
        /// <returns></returns>
        public IQueryable<TEntity>
GetWithInclude(System.Linq.Expressions.Expression<Func<TEntity, bool>> predicate, params
string[] include)
        {
            IQueryable<TEntity> query = this.DbSet;
            query = include.Aggregate(query, (current, inc) => current.Include(inc));
            return query.Where(predicate);
```

```
        }

        /// <summary>
        /// Generic method to check if entity exists
        /// </summary>
        /// <param name="primaryKey"></param>
        /// <returns></returns>
        public bool Exists(object primaryKey)
        {
            return DbSet.Find(primaryKey) != null;
        }

        /// <summary>
        /// Gets a single record by the specified criteria (usually the unique identifier)
        /// </summary>
        /// <param name="predicate">Criteria to match on</param>
        /// <returns>A single record that matches the specified criteria</returns>
        public TEntity GetSingle(Func<TEntity, bool> predicate)
        {
            return DbSet.Single<TEntity>(predicate);
        }

        /// <summary>
        /// The first record matching the specified criteria
        /// </summary>
        /// <param name="predicate">Criteria to match on</param>
        /// <returns>A single record containing the first record matching the specified
criteria</returns>
        public TEntity GetFirst(Func<TEntity, bool> predicate)
        {
            return DbSet.First<TEntity>(predicate);
        }


        #endregion
    }
}
```

Unit of Work :

Again I'll not explain in detail what Unit of Work is. You can google about the theory or follow my existing article on MVC with Unit of Work.

To give a heads up, again from my existing article, the important responsibilities of Unit of Work are,

- 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, different 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 business logic.

Step 1: Create a folder named UnitOfWork, add a class to that folder named UnitOfWork.cs,

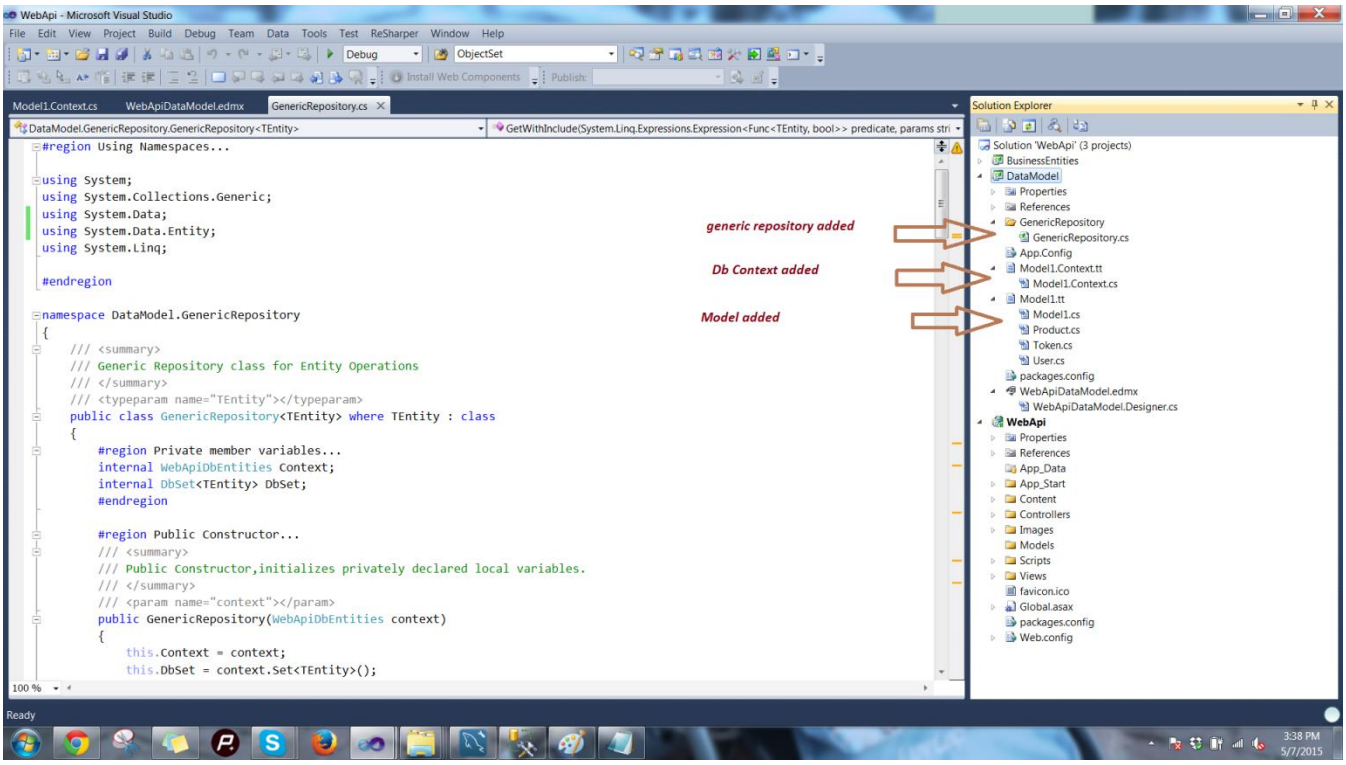
Add GenericRepository properties for all the three entities that we got. The class also implements IDisposable interface and it's method Dispose to free up connections and objects. The class will be as follows,

```
#region Using Namespaces...

using System;
using System.Collections.Generic;
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
    {
        #region Private member variables...

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

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

        #region Public Repository Creation properties...

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

        /// <summary>
```

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

        /// <summary>
        /// Get/Set Property for token repository.
        /// </summary>
        public GenericRepository<Token> TokenRepository
        {
            get
            {
                if (this._tokenRepository == null)
                    this._tokenRepository = new GenericRepository<Token>(_context);
                return _tokenRepository;
            }
        }
        #endregion

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

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

                throw e;
            }
```

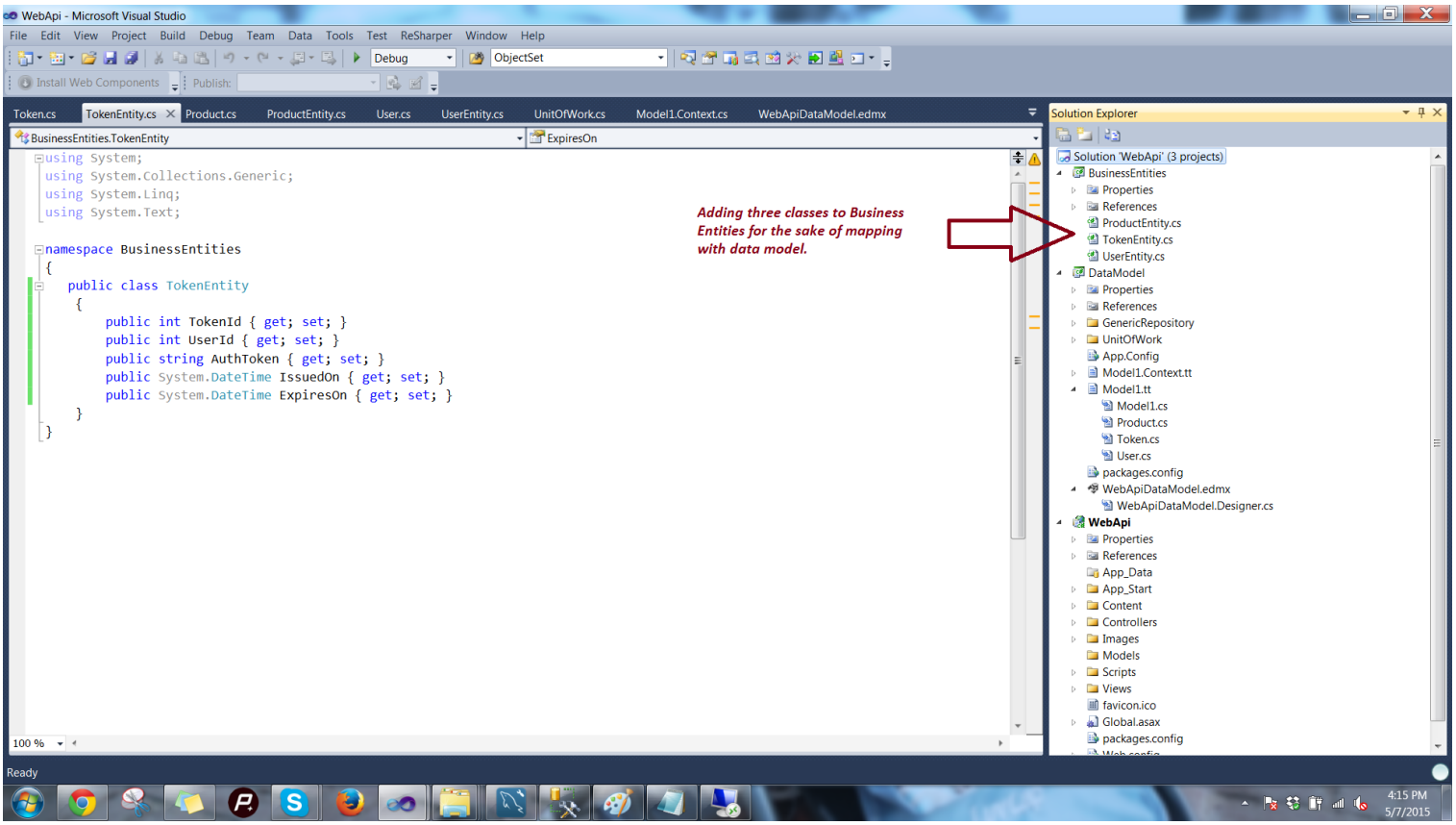
```
        }

        #endregion

        #region Implementing IDiosposable...

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

        /// <summary>
        /// Protected Virtual Dispose method
        /// </summary>
        /// <param name="disposing"></param>
        protected virtual void Dispose(bool disposing)
        {
            if (!this.disposed)
            {
                if (disposing)
                {
                    Debug.WriteLine("UnitOfWork is being disposed");
                    _context.Dispose();
                }
            }
            this.disposed = true;
        }

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

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



Setup Business Entities:

Remember, we created a business entities project. You may wonder, we already have database entities to interact with database then why do we need Business Entities?. The answer is as simple as that, 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 is Web API, it involves lot of risk.Hackers may manipulate the details and get accessto your database.Instead we'll use database entities in our business logic layer and use Business Entities as transfer objects to communicate between business logic and Web API project.So business entities may have different names but, their properties remains same as database entities. In our case we'll add same name business entity classes appendint word "Entity" to them in our BusinessEntity project. So we'll end up having three classes as follows,



Product entity :

```
public class ProductEntity
    {
        public int ProductId { get; set; }
        public string ProductName { get; set; }
    }
```

Token entity :

```
public class TokenEntity
    {
        public int TokenId { get; set; }
        public int UserId { get; set; }
        public string AuthToken { get; set; }
        public System.DateTime IssuedOn { get; set; }
        public System.DateTime ExpiresOn { get; set; }
    }
```

User entity :

```
public class UserEntity
    {
        public int UserId { get; set; }
```

```
        public string UserName { get; set; }
        public string Password { get; set; }
        public string Name { get; set; }
    }
```

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 make use of our API controllers to write business logic, but I am trying to segregate my business logic in an extra layer so that if in future I want to use WCF,MVC, Asp.net Web Pages or any other application as my presentation layer then I can easily integrate my Business logic layer in it.
We'll make this layer testable, so we need to create an interface in and declare CURD operations that we need to perform over product table.Before we proceed, add the reference of BusinessEntities project and DataModel project to this newly created project

Step 1: Create an interface named IProductServices and add following code to it for CURD operations methods,

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

namespace BusinessServices
{
    /// <summary>
    /// Product Service Contract
    /// </summary>
    public interface IProductServices
    {
        ProductEntity GetProductById(int productId);
        IEnumerable<ProductEntity> GetAllProducts();
        int CreateProduct(ProductEntity productEntity);
        bool UpdateProduct(int productId,ProductEntity productEntity);
        bool DeleteProduct(int productId);
    }
}
```

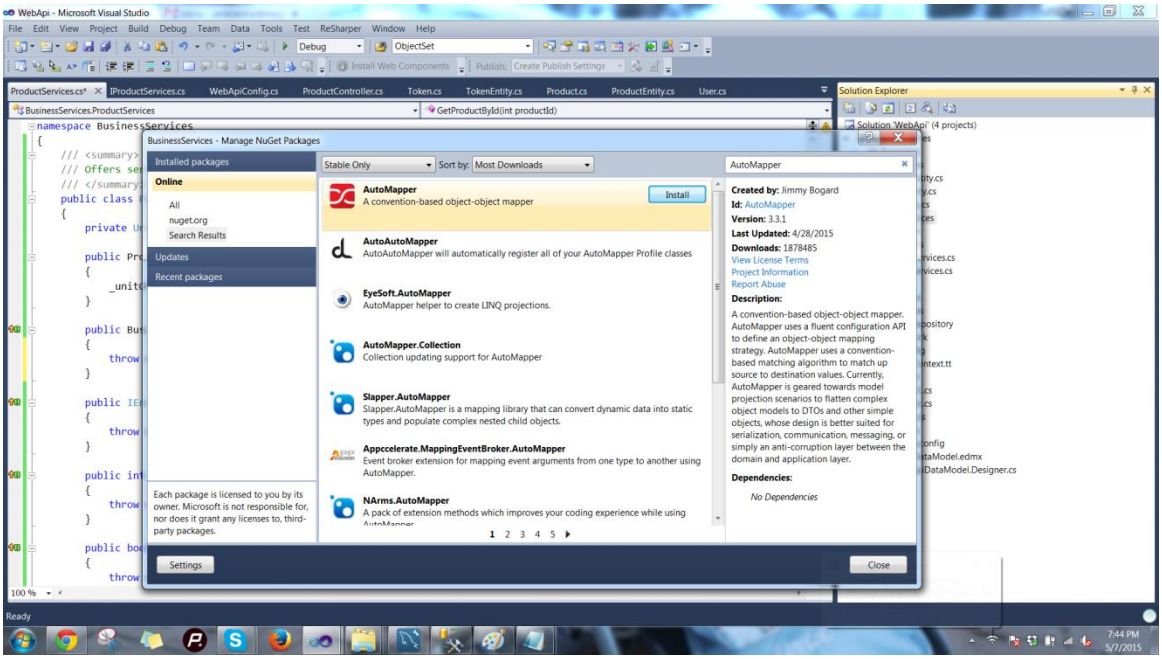
Step 2 : Create a class to implement this interface.name that class ProductServices,

The class contains a private variable of UnitOfWork and a constructor to initialize that variable,

```
      private readonly UnitOfWork _unitOfWork;

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

We have decided not to expose our db entities to Web API project, so we need something to map the db entities data to my business entity classes. We'll make use of AutoMapper.You can read about AutoMapper in my this article.

Step 3: Just right click project-> Extension manager, search for AutoMapper in online galary and add to BusinessServices project,



Step 4: Implement methods in ProductServices class ,

Add following code to the class,

```
using System.Collections.Generic;
using System.Linq;
using System.Transactions;
using AutoMapper;
using BusinessEntities;
using DataModel;
using DataModel.UnitOfWork;

namespace BusinessServices
{
    /// <summary>
    /// Offers services for product specific CRUD operations
    /// </summary>
    public class ProductServices:IProductServices
    {
        private readonly UnitOfWork _unitOfWork;

        /// <summary>
        /// Public constructor.
        /// </summary>
        public ProductServices()
```

```
{
    _unitOfWork = new UnitOfWork();
}

/// <summary>
/// Fetches product details by id
/// </summary>
/// <param name="productId"></param>
/// <returns></returns>
public BusinessEntities.ProductEntity GetProductById(int productId)
{
    var product = _unitOfWork.ProductRepository.GetByID(productId);
    if (product != null)
    {
        Mapper.CreateMap<Product, ProductEntity>();
        var productModel = Mapper.Map<Product, ProductEntity>(product);
        return productModel;
    }
    return null;
}

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

/// <summary>
/// Creates a product
/// </summary>
/// <param name="productEntity"></param>
/// <returns></returns>
public int CreateProduct(BusinessEntities.ProductEntity productEntity)
{
    using (var scope = new TransactionScope())
    {
        var product = new Product
        {
            ProductName = productEntity.ProductName
        };
        _unitOfWork.ProductRepository.Insert(product);
        _unitOfWork.Save();
        scope.Complete();
        return product.ProductId;
    }
}
```

```
        /// <summary>
        /// Updates a product
        /// </summary>
        /// <param name="productId"></param>
        /// <param name="productEntity"></param>
        /// <returns></returns>
        public bool UpdateProduct(int productId, BusinessEntities.ProductEntity
productEntity)
        {
            var success = false;
            if (productEntity != null)
            {
                using (var scope = new TransactionScope())
                {
                    var product = _unitOfWork.ProductRepository.GetByID(productId);
                    if (product != null)
                    {
                        product.ProductName = productEntity.ProductName;
                        _unitOfWork.ProductRepository.Update(product);
                        _unitOfWork.Save();
                        scope.Complete();
                        success = true;
                    }
                }
            }
            return success;
        }

        /// <summary>
        /// Deletes a particular product
        /// </summary>
        /// <param name="productId"></param>
        /// <returns></returns>
        public bool DeleteProduct(int productId)
        {
            var success = false;
            if (productId > 0)
            {
                using (var scope = new TransactionScope())
                {
                    var product = _unitOfWork.ProductRepository.GetByID(productId);
                    if (product != null)
                    {

                        _unitOfWork.ProductRepository.Delete(product);
                        _unitOfWork.Save();
                        scope.Complete();
                        success = true;
                    }
                }
            }
            return success;
        }
    }
}
```

Let me explain the idea of the code. We have 5 methods as follows,

1. To get product by id (GetproductById) : We call repository to get the product by id. Id comes as 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 calling method.

```
/// <summary>
/// Fetches product details by id
/// </summary>
/// <param name="productId"></param>
/// <returns></returns>
public BusinessEntities.ProductEntity GetProductById(int productId)
{
    var product = _unitOfWork.ProductRepository.GetByID(productId);
    if (product != null)
    {
        Mapper.CreateMap<Product, ProductEntity>();
        var productModel = Mapper.Map<Product, ProductEntity>(product);
        return productModel;
    }
    return null;
}
```

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

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

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

```
/// <summary>
/// Creates a product
/// </summary>
/// <param name="productEntity"></param>
/// <returns></returns>
public int CreateProduct(BusinessEntities.ProductEntity productEntity)
{
```

```
            using (var scope = new TransactionScope())
            {
                var product = new Product
                {
                    ProductName = productEntity.ProductName
                };
                _unitOfWork.ProductRepository.Insert(product);
                _unitOfWork.Save();
                scope.Complete();
                return product.ProductId;
            }
        }
```

I guess you can now write update and delete methods. So I am writing the code of complete class,

```
using System.Collections.Generic;
using System.Linq;
using System.Transactions;
using AutoMapper;
using BusinessEntities;
using DataModel;
using DataModel.UnitOfWork;

namespace BusinessServices
{
    /// <summary>
    /// Offers services for product specific CRUD operations
    /// </summary>
    public class ProductServices:IProductServices
    {
        private readonly UnitOfWork _unitOfWork;

        /// <summary>
        /// Public constructor.
        /// </summary>
        public ProductServices()
        {
            _unitOfWork = new UnitOfWork();
        }

        /// <summary>
        /// Fetches product details by id
        /// </summary>
        /// <param name="productId"></param>
        /// <returns></returns>
        public BusinessEntities.ProductEntity GetProductById(int productId)
        {
            var product = _unitOfWork.ProductRepository.GetByID(productId);
            if (product != null)
            {
                Mapper.CreateMap<Product, ProductEntity>();
                var productModel = Mapper.Map<Product, ProductEntity>(product);
                return productModel;
            }
            return null;
```

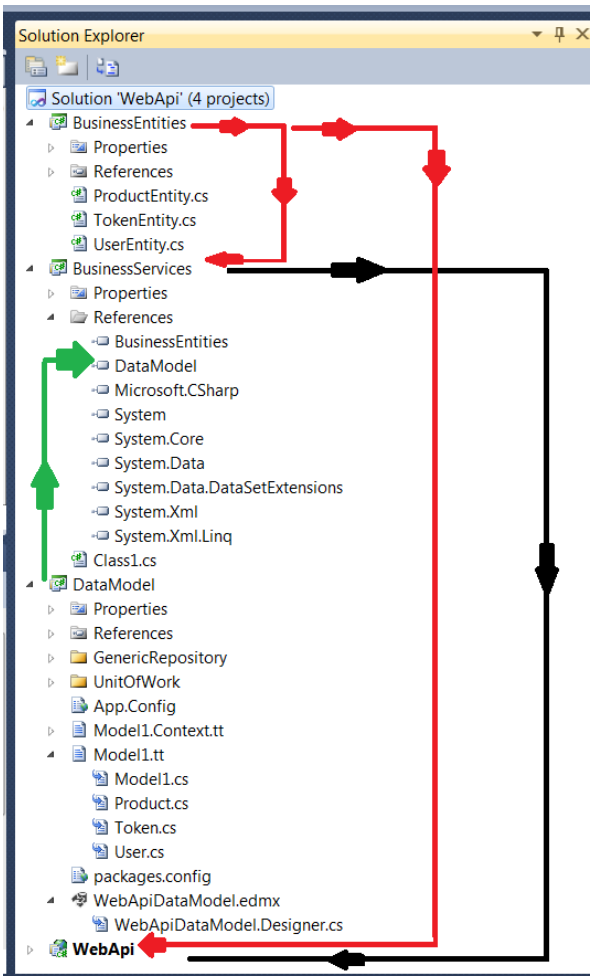
```
        }

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

        /// <summary>
        /// Creates a product
        /// </summary>
        /// <param name="productEntity"></param>
        /// <returns></returns>
        public int CreateProduct(BusinessEntities.ProductEntity productEntity)
        {
            using (var scope = new TransactionScope())
            {
                var product = new Product
                {
                    ProductName = productEntity.ProductName
                };
                _unitOfWork.ProductRepository.Insert(product);
                _unitOfWork.Save();
                scope.Complete();
                return product.ProductId;
            }
        }

        /// <summary>
        /// Updates a product
        /// </summary>
        /// <param name="productId"></param>
        /// <param name="productEntity"></param>
        /// <returns></returns>
        public bool UpdateProduct(int productId, BusinessEntities.ProductEntity
productEntity)
        {
            var success = false;
            if (productEntity != null)
            {
                using (var scope = new TransactionScope())
                {
                    var product = _unitOfWork.ProductRepository.GetByID(productId);
                    if (product != null)
                    {
                        product.ProductName = productEntity.ProductName;
```

```
                    _unitOfWork.ProductRepository.Update(product);
                    _unitOfWork.Save();
                    scope.Complete();
                    success = true;
                }
            }
        }
        return success;
    }

    /// <summary>
    /// Deletes a particular product
    /// </summary>
    /// <param name="productId"></param>
    /// <returns></returns>
    public bool DeleteProduct(int productId)
    {
        var success = false;
        if (productId > 0)
        {
            using (var scope = new TransactionScope())
            {
                var product = _unitOfWork.ProductRepository.GetByID(productId);
                if (product != null)
                {

                    _unitOfWork.ProductRepository.Delete(product);
                    _unitOfWork.Save();
                    scope.Complete();
                    success = true;
                }
            }
        }
        return success;
    }
  }
}
```

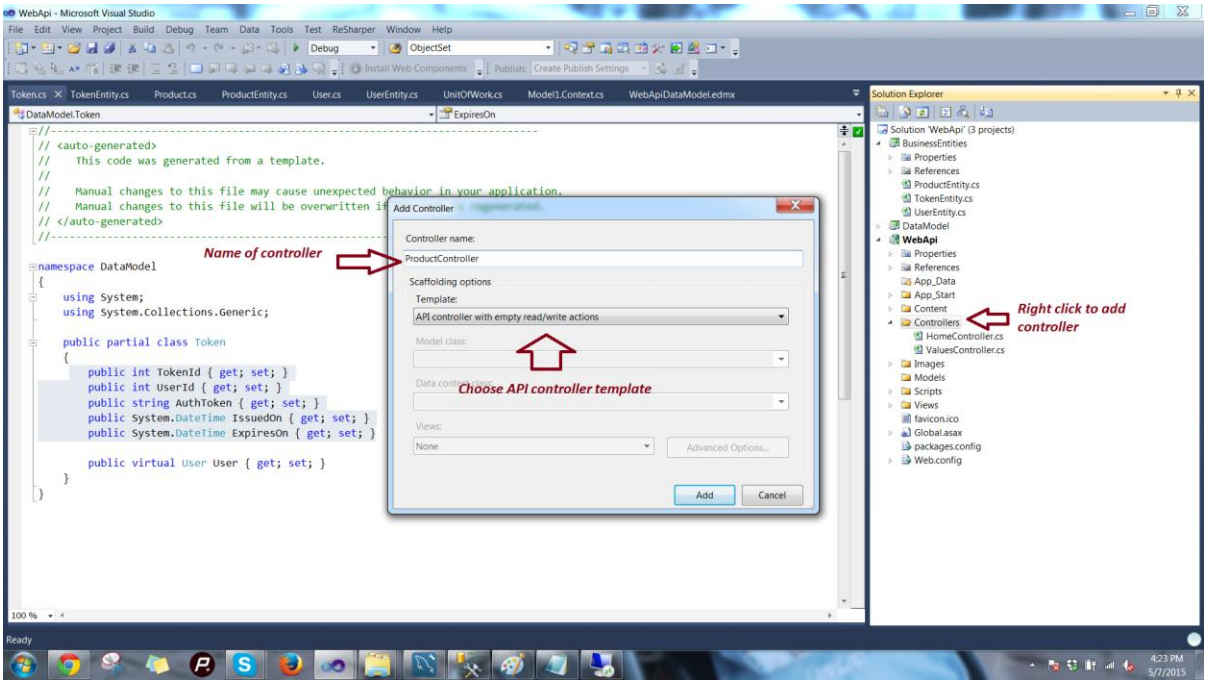
Job done at business service level.Let's move on to API controller to call these methods.

Setup WebAPI project:

Step1 :
Just add the reference of BusinessEntity and BusinessService in the WebAPI project, our architecture becomes like this,



Step 2: Add a new WebAPI controller in Controller folder.Right click Controller folder and add a new controller.



We get a controller as follows,

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace WebApi.Controllers
{
    public class ProductController : ApiController
    {
        // GET api/product
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/product/5
        public string Get(int id)
        {
            return "value";
        }

        // POST api/product
        public void Post([FromBody]string value)
        {
        }

        // PUT api/product/5
        public void Put(int id, [FromBody]string value)
        {
        }

        // DELETE api/product/5
        public void Delete(int id)
        {
        }
    }
}
```

We get HTTP VERBS as method names. Web API is smart enough to recognize 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 have to write calling logic inside these methods. In my upcoming articles of the series, we will figure out how we can define new routes and provide method names of our choice with those routes.

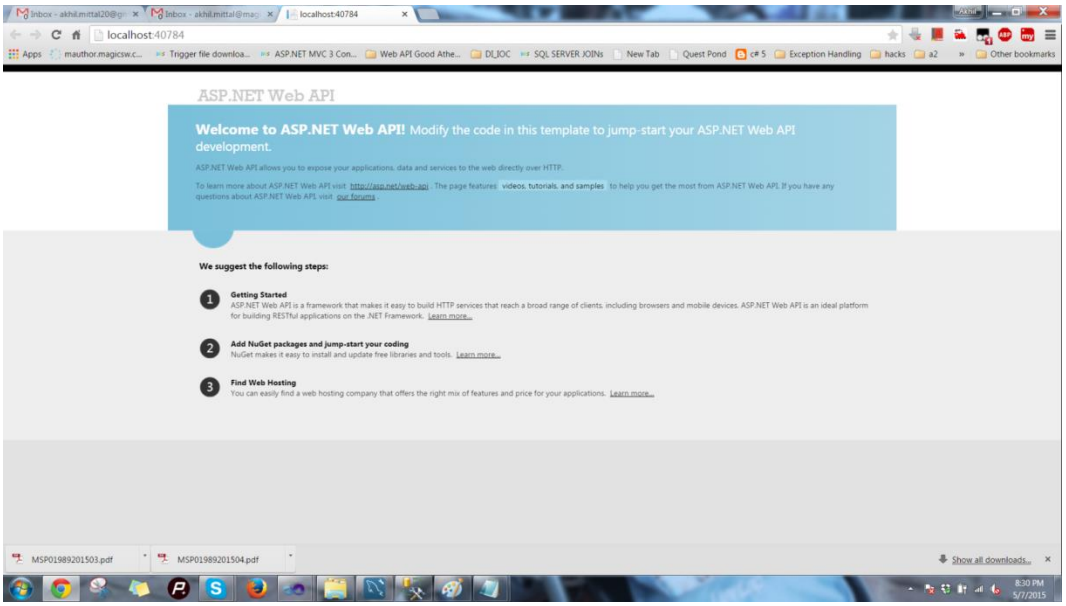
Step 3: Add logic to call Business Service methods, just make an object of Business Service and call its respective methods, our Controller class becomes like,

```
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()
        {
            _productServices =new 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)
            {
```

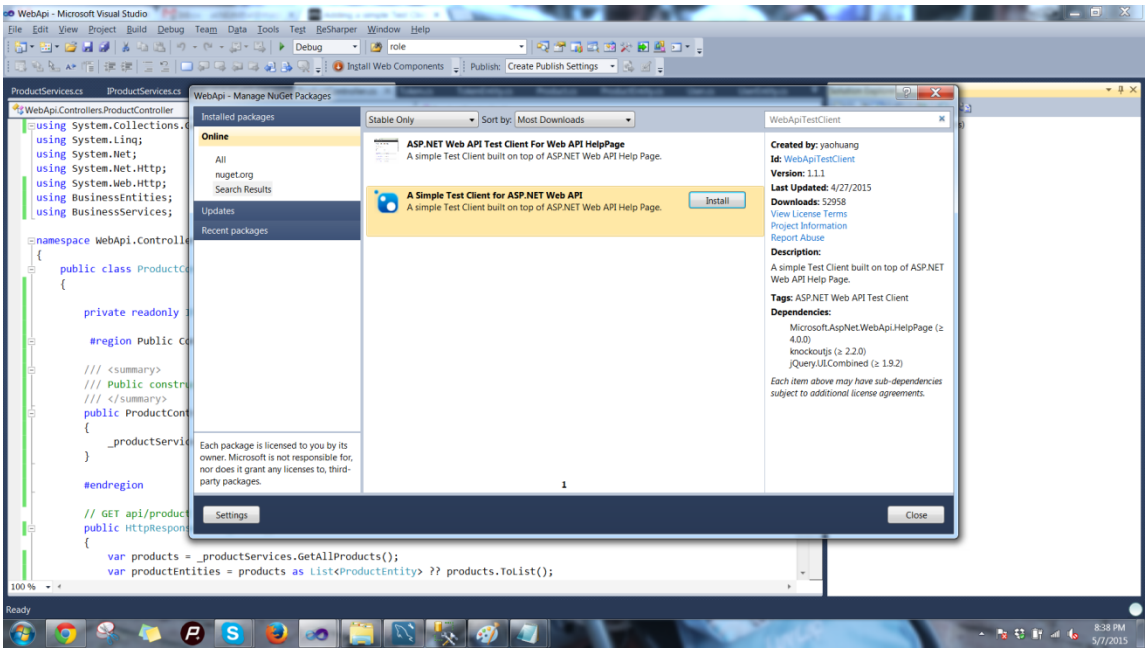
```
                return _productServices.UpdateProduct(id, productEntity);
            }
            return false;
        }

        // DELETE api/product/5
        public bool Delete(int id)
        {
            if (id > 0)
                return _productServices.DeleteProduct(id);
            return false;
        }
    }
}
```

Just run the application, we get,

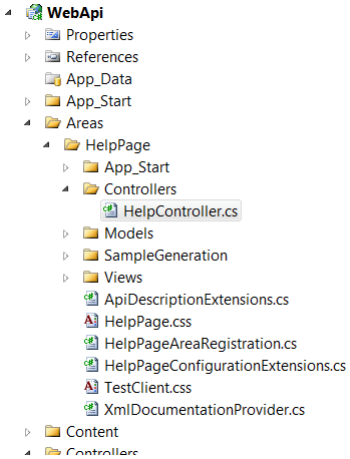


But now how do we test our API? We don't have client. Guys, we'll not be writing a client now to test it.We'll add a package that will do all our work.

Just go to Manage Nuget Packages, by right clicking WebAPI project and type WebAPITestClient in searchbox in online packages,



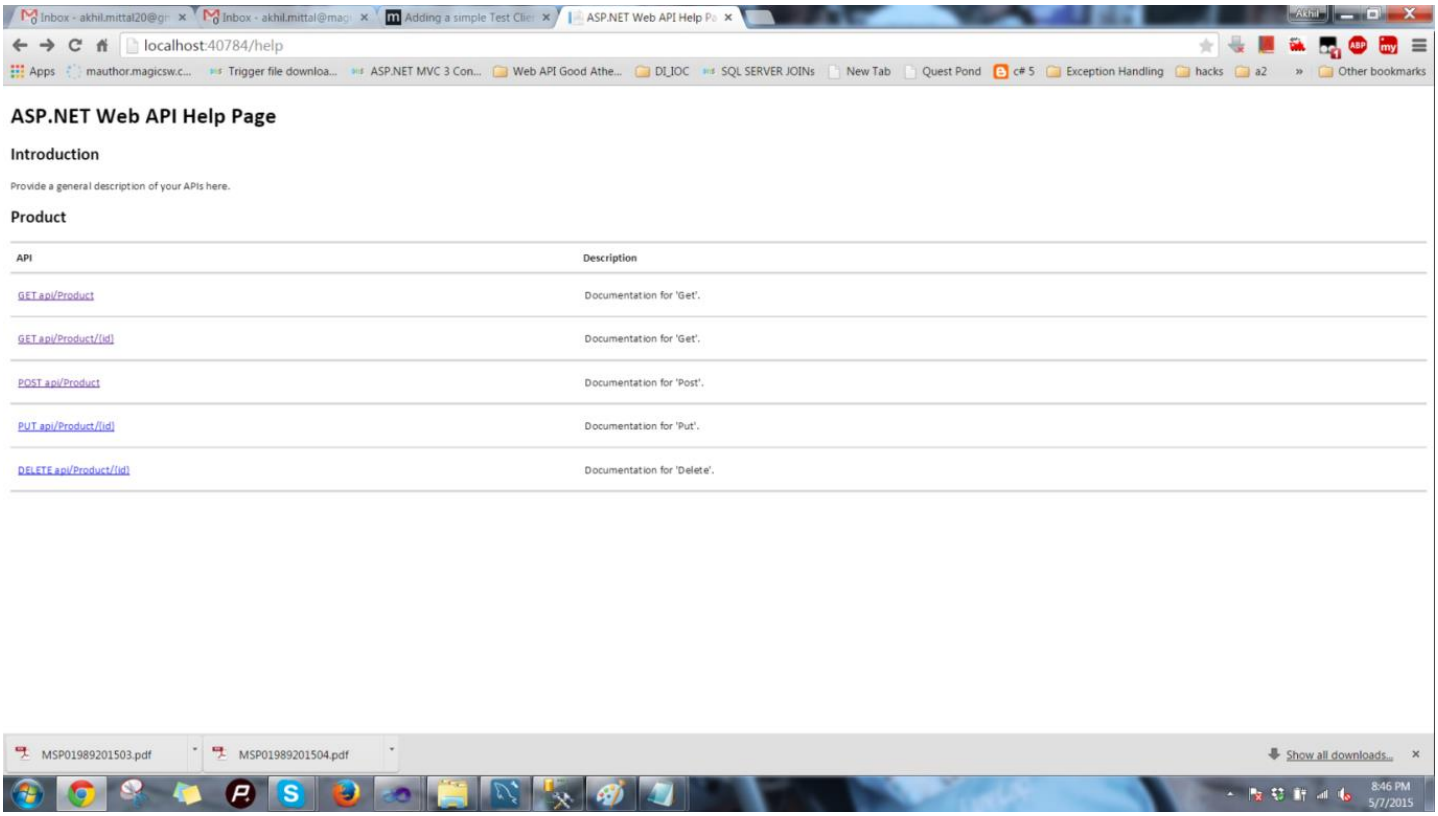
You'll get "A simple Test Client for ASP.NET Web API", just add it. You'll get a help controller in Areas-> HelpPage like shown below,



Running the Application:

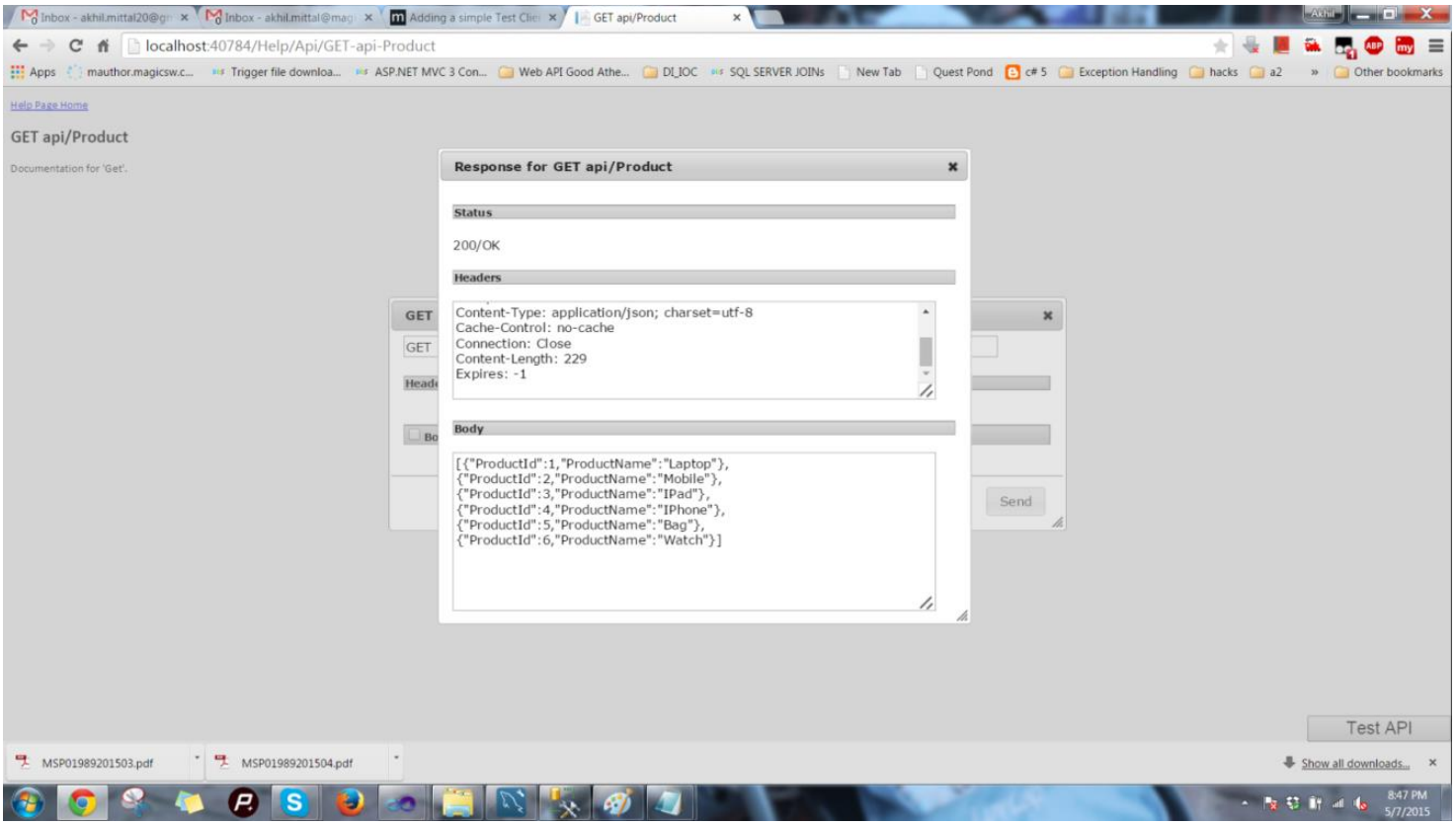
Before running the application, I have put some test data in our product table.

Just hit F5, you get the same page as you got earlier, just append "/help" in its url, and you'll get the test client,

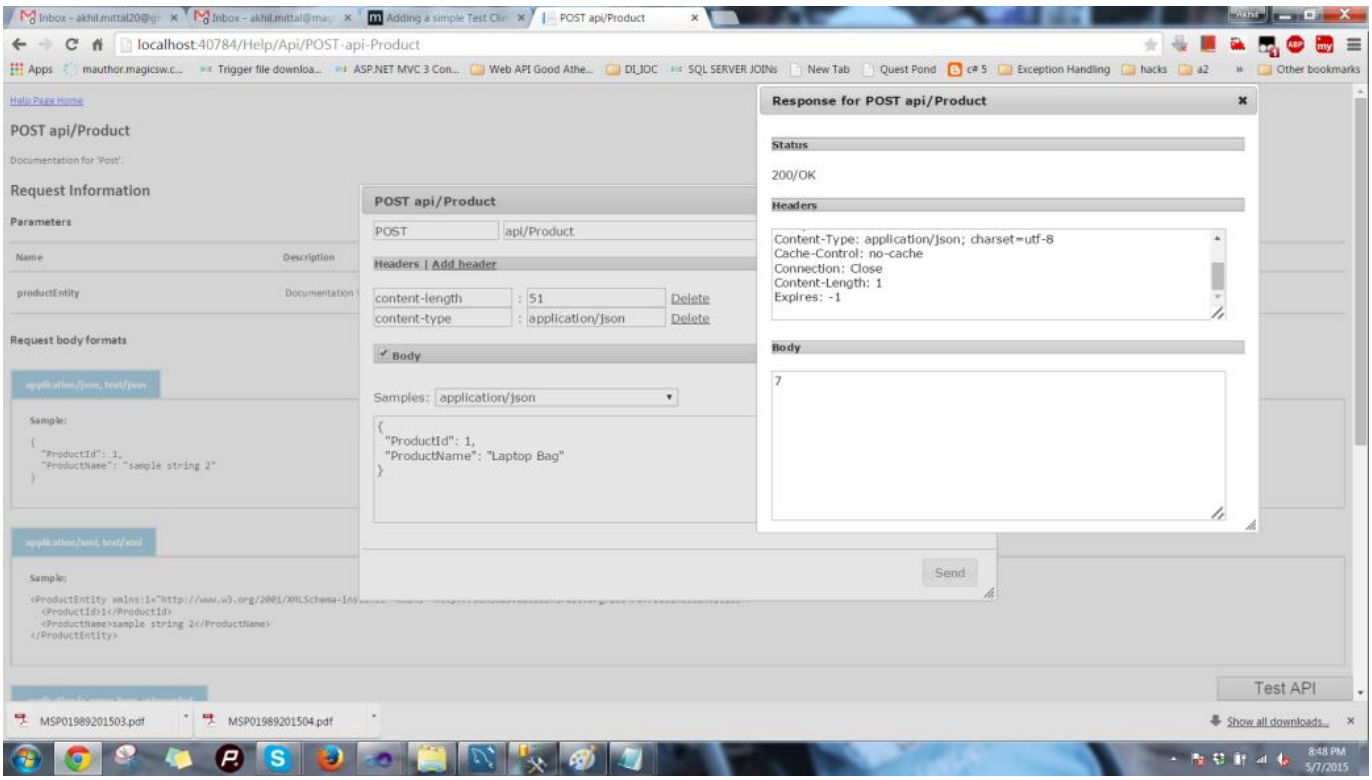


You can test each service by clicking on it.

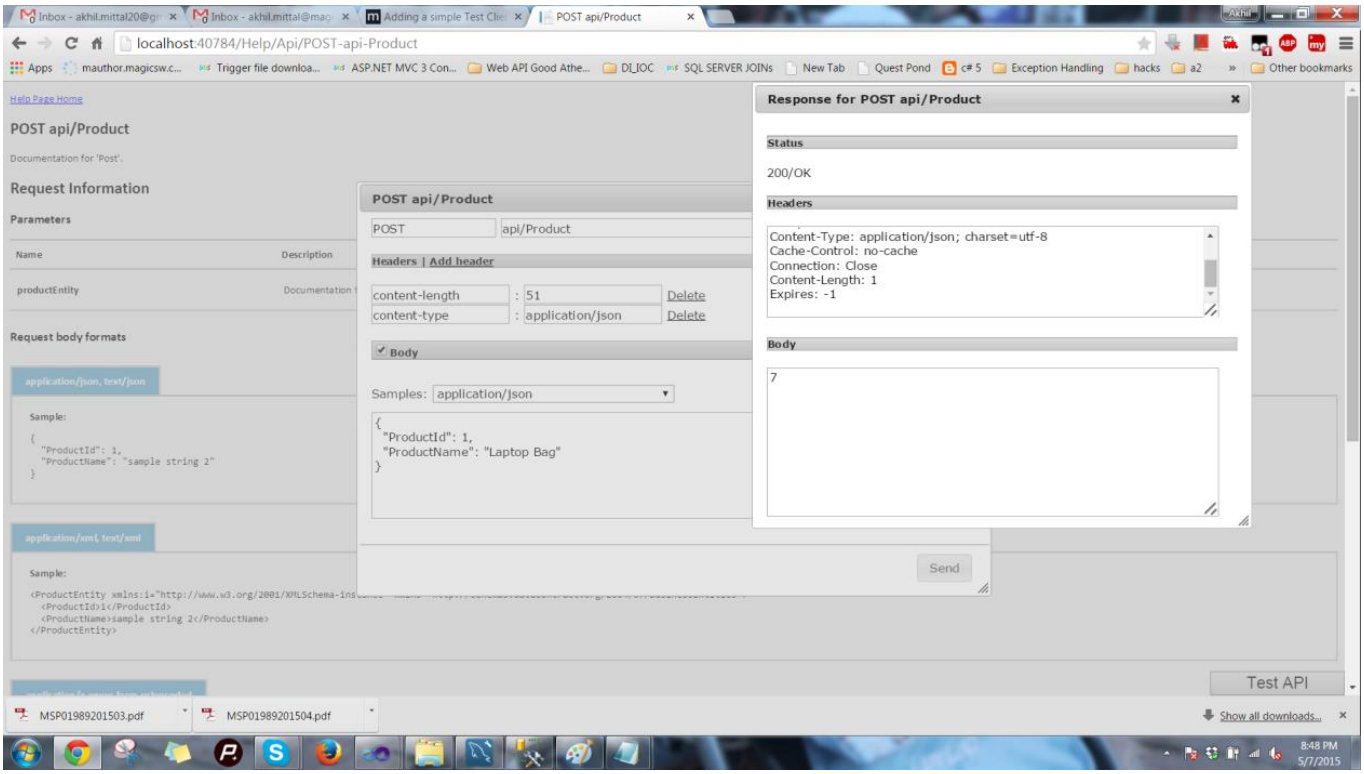
Service for GetAllProduct,



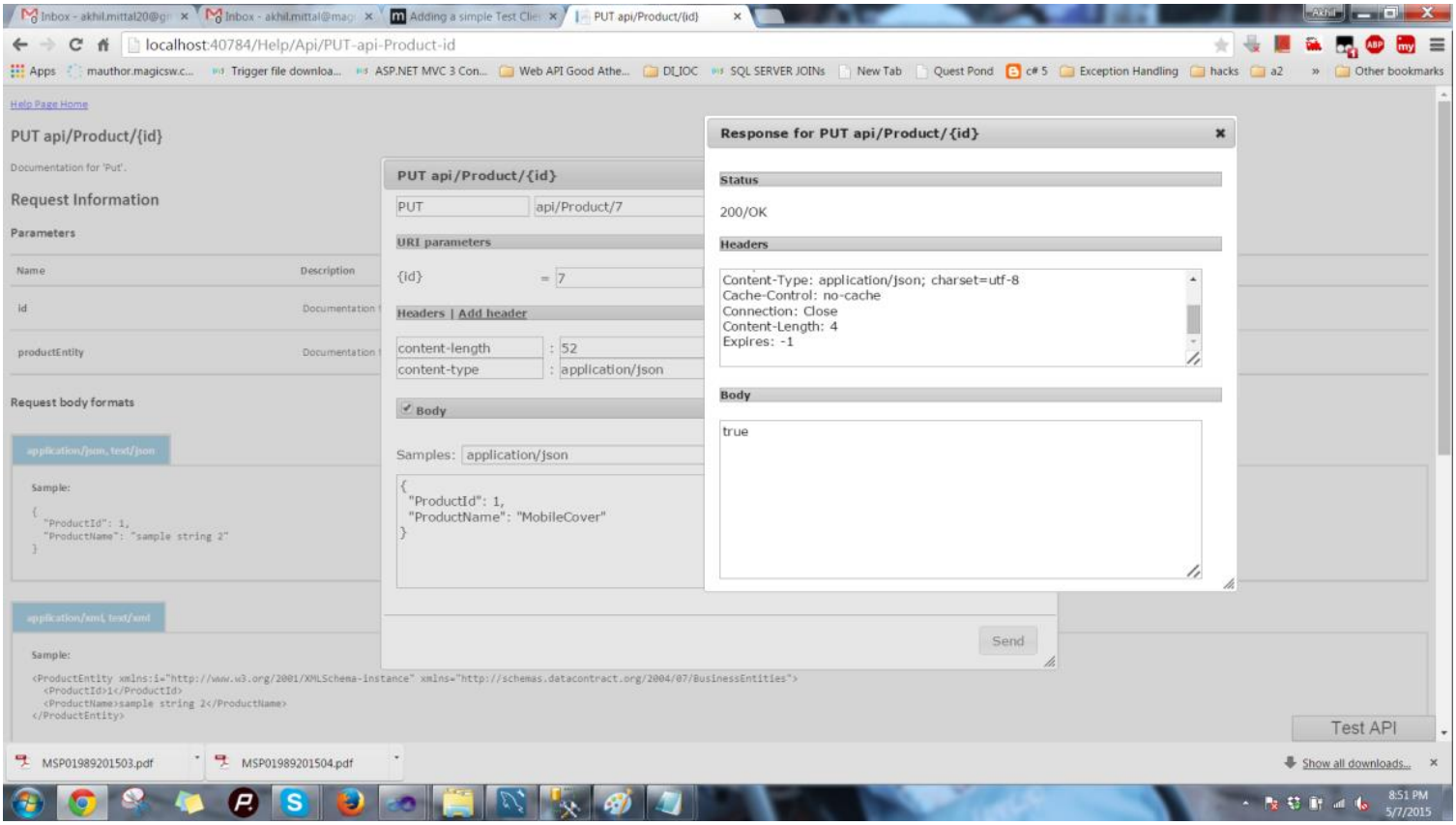
For Create a new product,



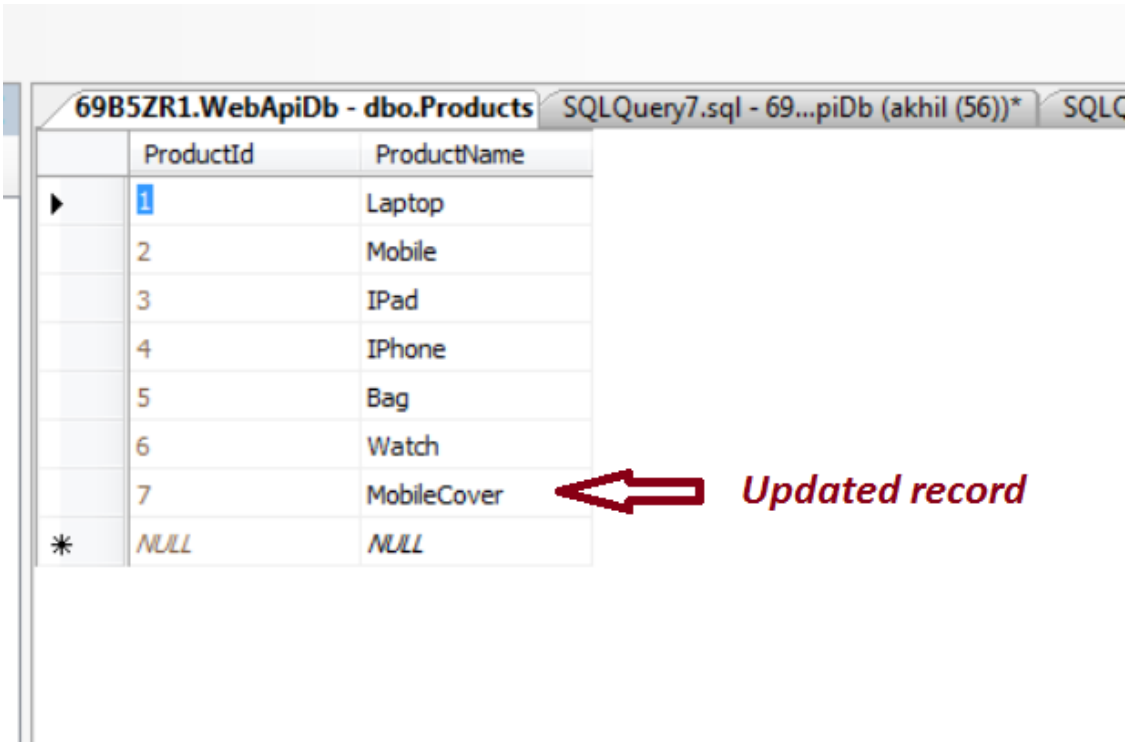
In database, we get new product,



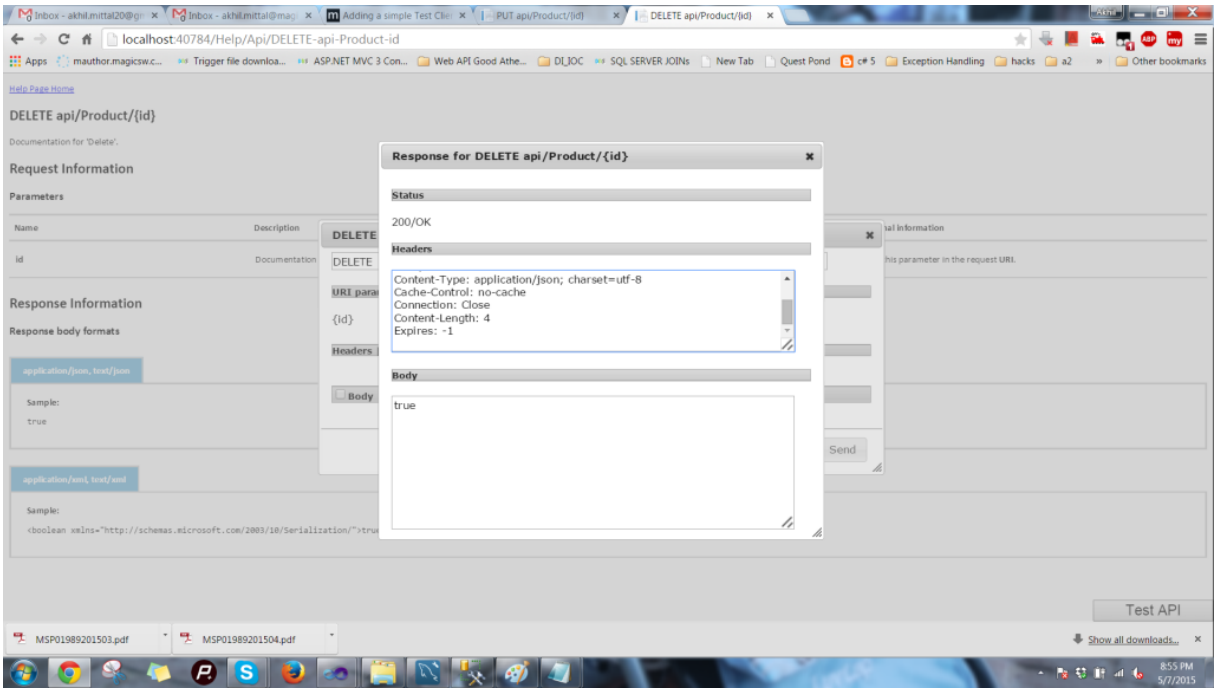
Update product:



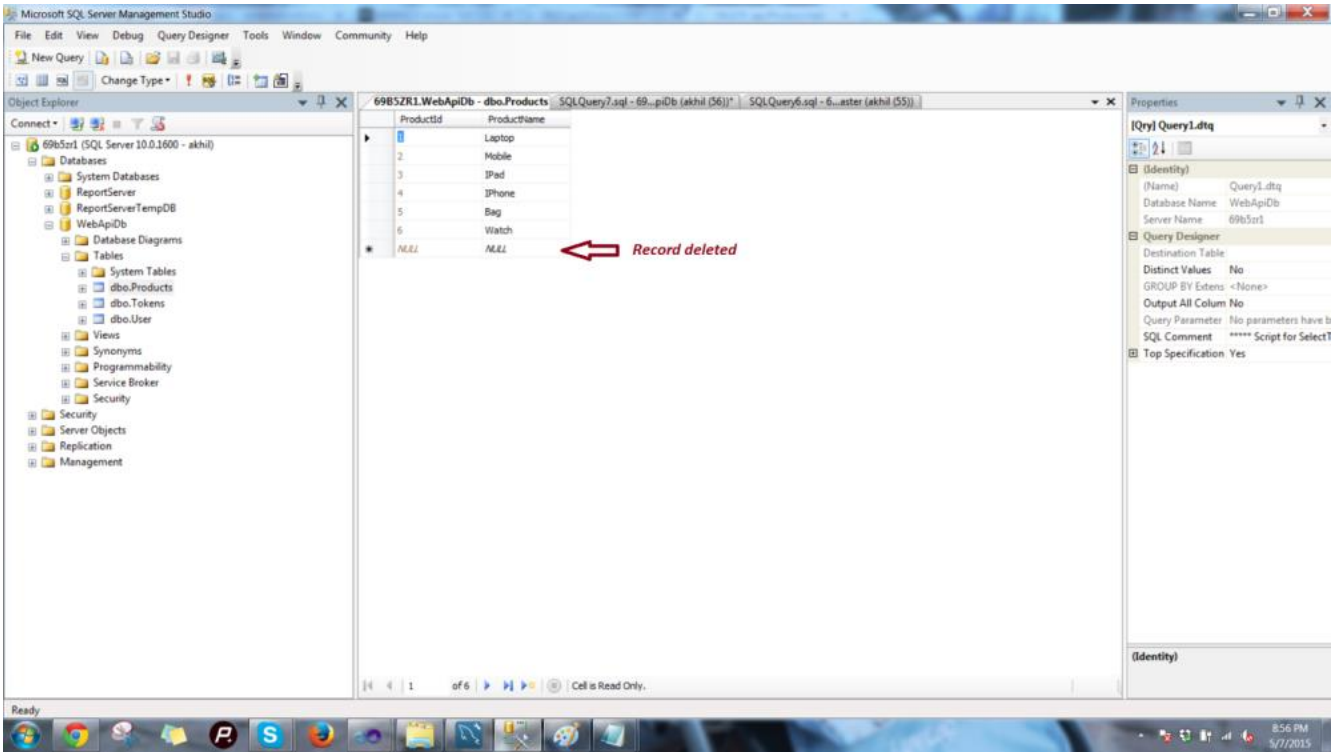
We get in database,



Delete product:



In database:



Job done.

Design Flaws

1. Architecture is tightly coupled. IOC (Inversion of Control) 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 n layered architecture.
But still there are some flaws in this design.In my next two articles I'll explain how to make the system loosely coupled using Dependency Injection Principle. We'll also cover all the design flaws to make our design better and stronger. Till then Happy Coding ☺ You can also download the source code from GitHub.

About Author

Akhil Mittal works as a Sr. Analyst in **Magic Software** and have an experience of more than 8 years in C#.Net. He is a codeproject and a c-sharpcorner MVP (Most Valuable Professional). Akhil is a B.Tech (Bachelor of Technology) in Computer Science and holds a diploma in Information Security and Application Development from CDAC. His work experience includes Development of Enterprise Applications using C#, .Net and Sql Server, Analysis as well as Research and Development. His expertise is in web application development. He is a MCP (Microsoft Certified Professional) in Web Applications (MCTS-70-528, MCTS-70-515) and .Net Framework 2.0 (MCTS-70-536).