RESTful Day #2: Inversion of control using dependency injection in Web API's using Unity Container and Bootstrapper

–by Akhil Mittal

Table of Contents

Introduction:

My article will explain how we can make our Web API service architecture loosely coupled and more flexible. We already learnt that how we can create a RESTful service using Asp.net Web API and Entity framework in my last article. If you remember we ended up in a solution with a design flaw, we'll try to overcome that flaw by resolving the dependencies of dependent components. For those who have not followed my previous article, they can learn by having the sample project attached as a test application from my first article.

Image source : https://www.pehub.com/wp-content/uploads/2013/06/independent-300x200.jpg

There are various methods you can use to resolve dependency of components. In my article I'll explain how to resolve dependency with the help of Unity Container provided by Microsoft's Unity Application Block.
We'll not go into very detailed theory, for theory and understanding of DI and IOC you can follow the following links: Unity and Inversion of Control(IOC).We'll straight away jump into practical implementation.

Roadmap:

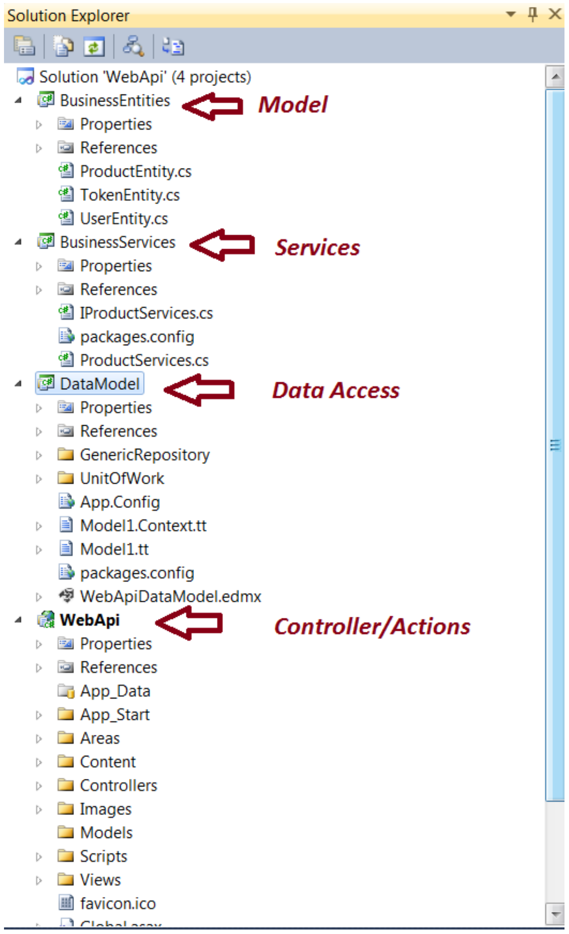
Our roadmap for learning RESTful APIs remains same,

- 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.

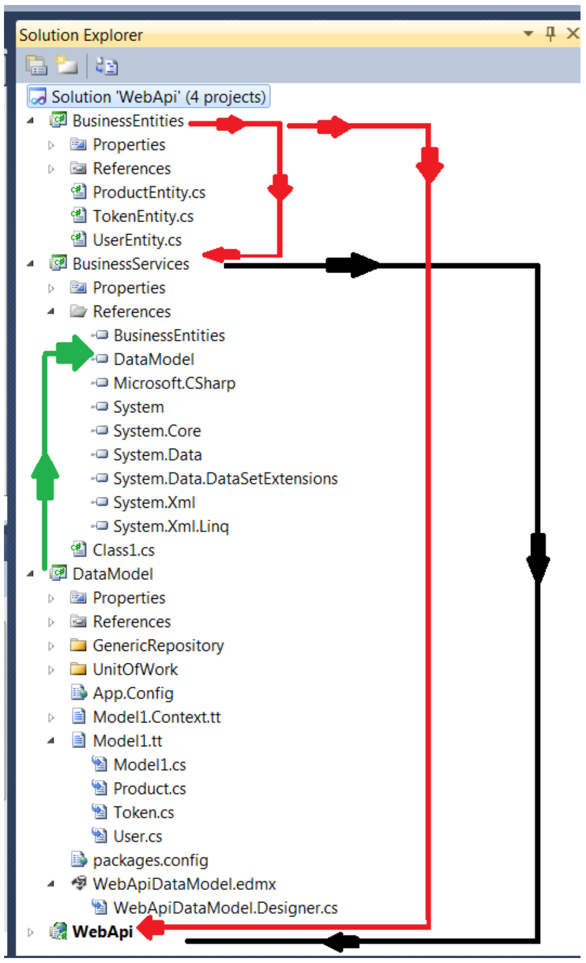
I'll purposely use Visual Studio 2010 and .net Framework 4.0 because there are few implementations that are very hard to find in .Net Framework 4.0, but I'll make it easy by showing how we can do it.

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,



The modules are dependent in a way,



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

For e.g.

Controller constructor makes an object of Service layer to communicate,

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

Service constructor in turn makes and object of UnitOfWork to communicate to database,

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

The problem lies in these code pieces. We see Controller is dependent upon instantiation of Service and Service is dependent upon UnitOfWork to get instantiated. Our Layers should not be that tightly coupled and should be dependant to each other.
The work of creating object should be assigned to someone else. Our layers should not worry about creating objects. We'll assign this role to a third party that will be called our container. Fortunately Unity provides that help to us, to get rid of this dependency problem and invert the control flow by injecting dependency not by creating objects by new but through 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,

Image source: http://img.tradeindia.com/fp/1/669/664.jpg

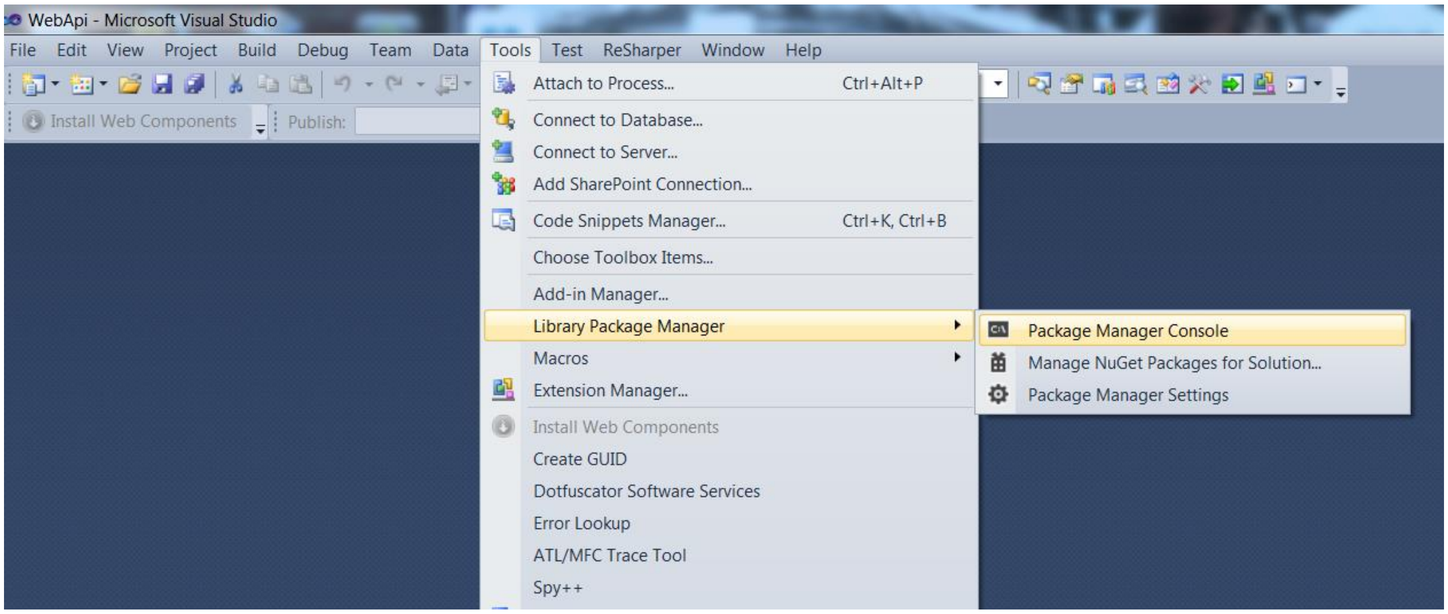
"The Unity Application Block (Unity) is a lightweight, extensible dependency injection container that supports constructor injection, property injection, and method call injection. It provides developers with the following advantages:

- *It provides simplified object creation, especially for hierarchical object structures and dependencies, which 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:

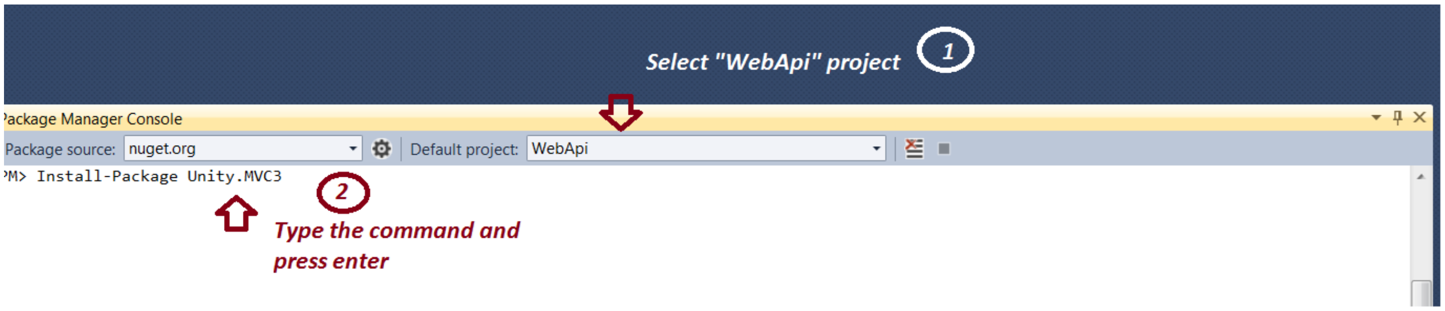
Open your Visual Studio , I am using VS 2010, You can use VS version 2010 or above. Load the solution.

Step 1: browse to Tools-> Library Packet Manager - > Packet manager Console,



We'll add package for Unity Application Block.
In the left bottom 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.

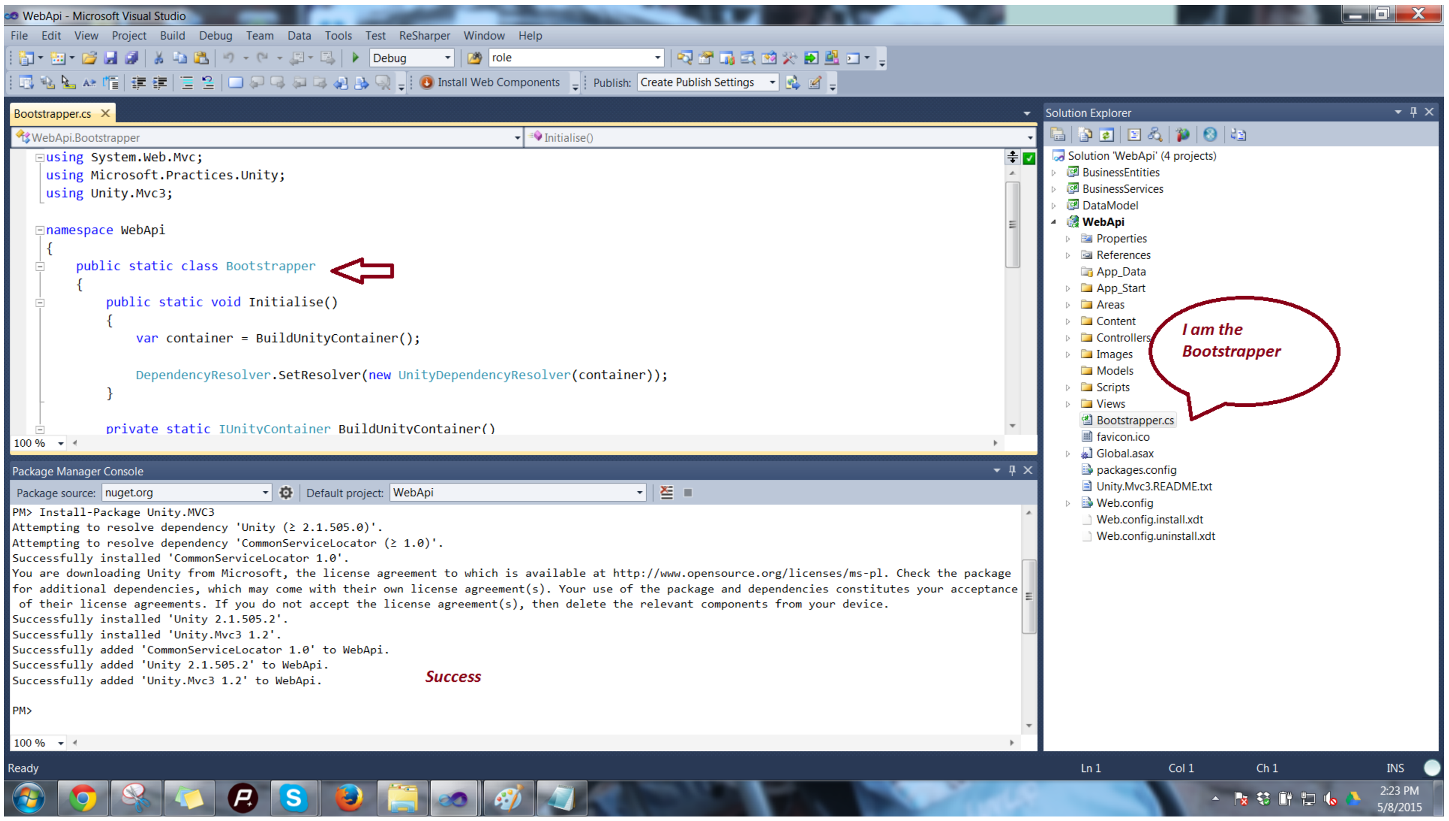


```
PM> Install-Package Unity.MVC3
```

Step 2 : Bootstrapper class

Unity.MVC3 comes with a Bootstrapper class, as soon as you run the command, the Bootstrapper class will be generated in your solution->WebAPI project,

```
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;
        }
    }
}
```



```
using System.Web.Mvc;
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));
        }

        private static IUnityContainer BuildUnityContainer()
```

```
PM> Install-Package Unity.MVC3
Attempting to resolve dependency 'Unity (≥ 2.1.505.0)'.
Attempting to resolve dependency 'CommonServiceLocator (≥ 1.0)'.
Successfully installed 'CommonServiceLocator 1.0'.
You are downloading Unity from Microsoft, the license agreement to which is available at http://www.opensource.org/licenses/ms-pl. Check the package
for additional dependencies, which may come with their own license agreement(s). Your use of the package and dependencies constitutes your acceptance
 of their license agreements. If you do not accept the license agreement(s), then delete the relevant components from your device.
Successfully installed 'Unity 2.1.505.2'.
Successfully installed 'Unity.Mvc3 1.2'.
Successfully added 'CommonServiceLocator 1.0' to WebApi.
Successfully added 'Unity 2.1.505.2' to WebApi.
Successfully added 'Unity.Mvc3 1.2' to WebApi.

PM>
```

This class comes with an initial configuration to setup your container. All the functionality is inbuilt, we only need to specify the dependencies that we need to resolve in the "BuildUnityContainer", like it says in the commented statement,

```
            // register all your components with the container here
            // it is NOT necessary to register your controllers

            // 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,

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

"`HierarchicalLifetimeManager`" maintains the lifetime of the object and child object depends upon parent object's lifetime.
If you don't find "`UnitOfWork`", just add reference to DataModel project in WebAPI project.

So our Bootstrapper class becomes,

```
   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;
        }
```

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

Step 4 : Now we need to call the Initialise method of Bootstrapper class. Note , we need the objects as soon as our modules load, therefore we require the container to do its work at the time of application load, therefore go to Global.asax file and add one line to call Initialise method, since this is a static method, we can directly call it using class name,

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

Our global.asax becomes,

```
using System.Linq;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using Newtonsoft.Json;
using WebApi.App_Start;

namespace WebApi
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit http://go.microsoft.com/?LinkId=9394801

    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            //Initialise Bootstrapper
            Bootstrapper.Initialise();

            //Define Formatters
            var formatters = GlobalConfiguration.Configuration.Formatters;
            var jsonFormatter = formatters.JsonFormatter;
            var settings = jsonFormatter.SerializerSettings;
            settings.Formatting = Formatting.Indented;
            // settings.ContractResolver = new CamelCasePropertyNamesContractResolver();
            var appXmlType = formatters.XmlFormatter.SupportedMediaTypes.FirstOrDefault(t =>
t.MediaType == "application/xml");
            formatters.XmlFormatter.SupportedMediaTypes.Remove(appXmlType);

            //Add CORS Handler
            GlobalConfiguration.Configuration.MessageHandlers.Add(new CorsHandler());
        }
    }
}
```

Half of the job is done.We now need to touchbase 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 dependency, like constructor injection, property injection, via service locator. I am here using Constructor Injection, because I find it best method to use with Unity Container to resolve dependency.

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

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

Just add a parameter to your constructor that takes your ProductServices reference, like we did below

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

And initialize your “`productServices`” variable with the parameter. In this case when the constructor of the controller is called, It will be served with 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 same fashion. Just open your ProductServices class, we see the dependency of UnitOfWork here as,

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

Again, we perform the same steps ,and pass a parameter of type UnitOfWork to our constructor,
Our code becomes,

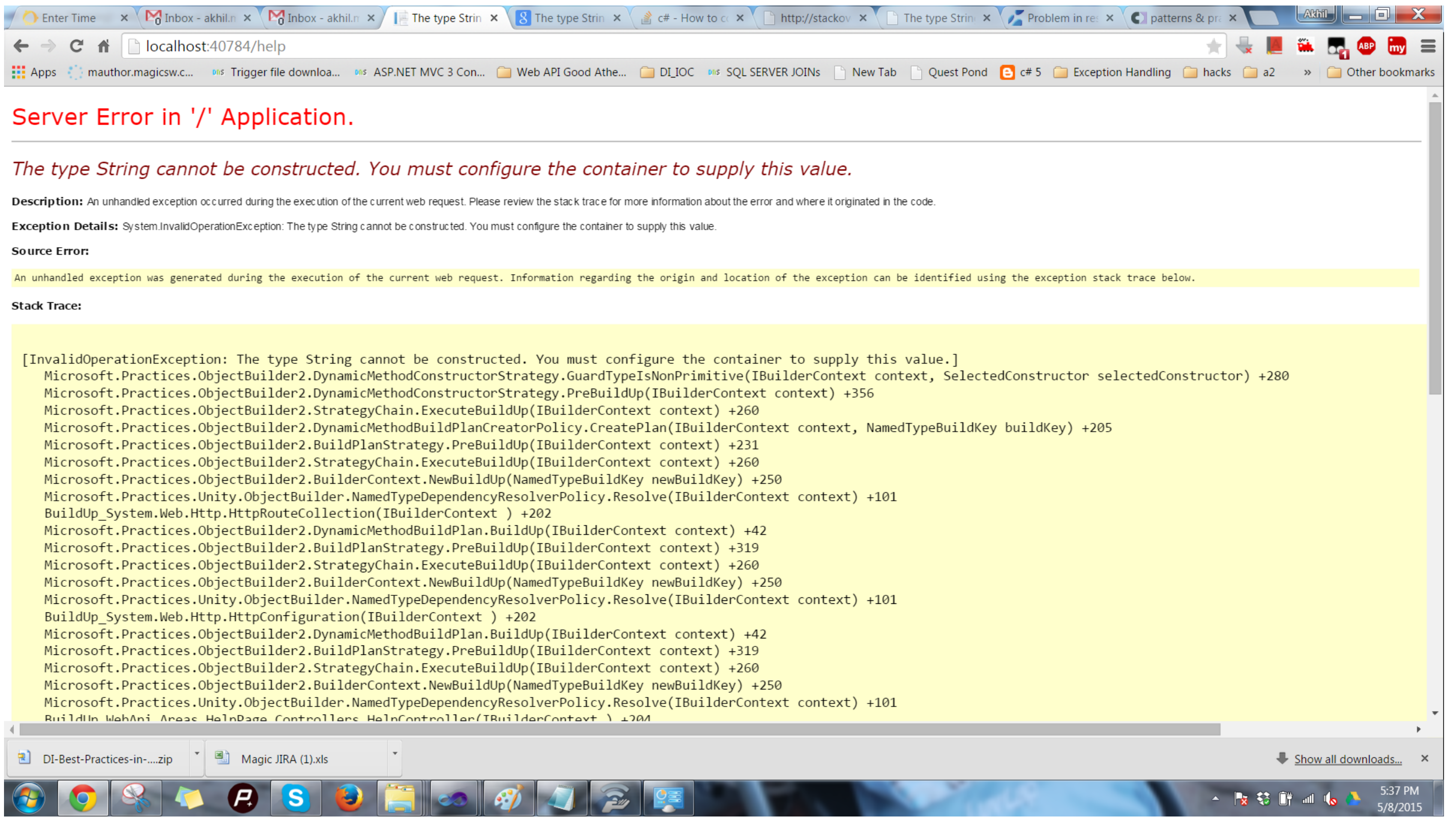
```
        /// <summary>
        /// Public constructor.
        /// </summary>
        public ProductServices(UnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }
```

Here also we’ll get the pre instantiated object on UnitOfWork.So service does need to worry about creating objects.Remember we did `.RegisterType<UnitOfWork>() in Bootstrapper class.`
We have now made our components independent.

Image source: http://4.bp.blogspot.com/-q-o5SXbf3jw/T0ZUv0vDafI/AAAAAAAAAY4/_O8PgPNXIKQ/s320/h1.jpg

Running the application :

Our job is almost done.We need to run the application, Just hit F5. To our surprise we'll end up in an error page,



Server Error in '/' Application.

The type String cannot be constructed. You must configure the container to supply this value.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.InvalidOperationException: The type String cannot be constructed. You must configure the container to supply this value.

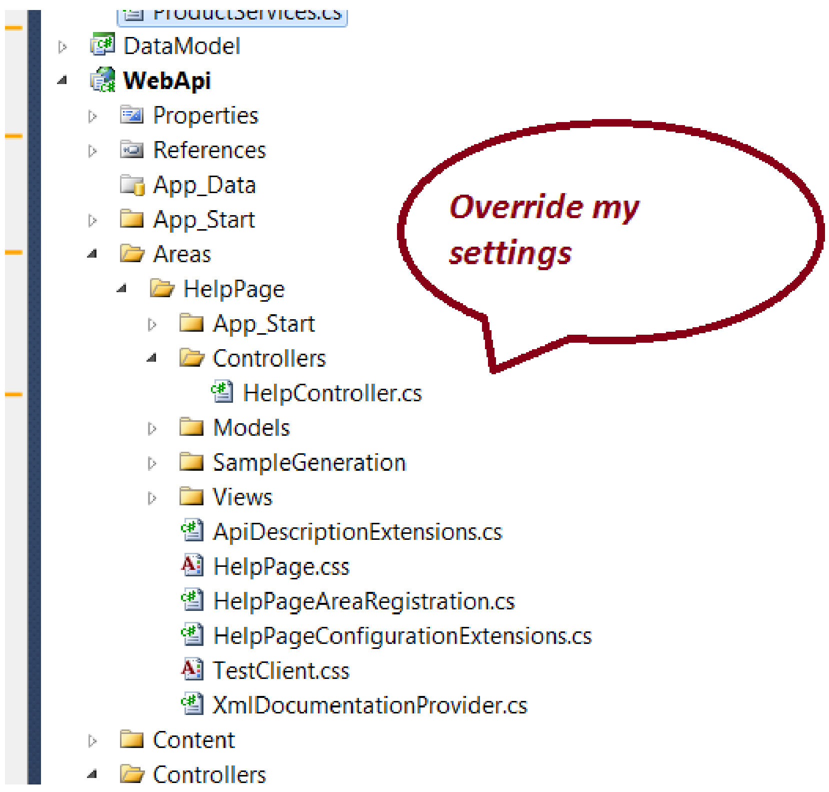
Source Error:

```
An unhandled exception was generated during the execution of the current web request. Information regarding the origin and location of the exception can be identified using the exception stack trace below.
```

Stack Trace:

```
[InvalidOperationException: The type String cannot be constructed. You must configure the container to supply this value.]
   Microsoft.Practices.ObjectBuilder2.DynamicMethodConstructorStrategy.GuardTypeIsNonPrimitive(IBuilderContext context, SelectedConstructor selectedConstructor) +280
   Microsoft.Practices.ObjectBuilder2.DynamicMethodConstructorStrategy.PreBuildUp(IBuilderContext context) +356
   Microsoft.Practices.ObjectBuilder2.StrategyChain.ExecuteBuildUp(IBuilderContext context) +260
   Microsoft.Practices.ObjectBuilder2.DynamicMethodBuildPlanCreatorPolicy.CreatePlan(IBuilderContext context, NamedTypeBuildKey buildKey) +205
   Microsoft.Practices.ObjectBuilder2.BuildPlanStrategy.PreBuildUp(IBuilderContext context) +231
   Microsoft.Practices.ObjectBuilder2.StrategyChain.ExecuteBuildUp(IBuilderContext context) +260
   Microsoft.Practices.ObjectBuilder2.BuilderContext.NewBuildUp(NamedTypeBuildKey newBuildKey) +250
   Microsoft.Practices.Unity.ObjectBuilder.NamedTypeDependencyResolverPolicy.Resolve(IBuilderContext context) +101
   BuildUp_System.Web.Http.HttpRouteCollection(IBuilderContext ) +202
   Microsoft.Practices.ObjectBuilder2.DynamicMethodBuildPlan.BuildUp(IBuilderContext context) +42
   Microsoft.Practices.ObjectBuilder2.BuildPlanStrategy.PreBuildUp(IBuilderContext context) +319
   Microsoft.Practices.ObjectBuilder2.StrategyChain.ExecuteBuildUp(IBuilderContext context) +260
   Microsoft.Practices.ObjectBuilder2.BuilderContext.NewBuildUp(NamedTypeBuildKey newBuildKey) +250
   Microsoft.Practices.Unity.ObjectBuilder.NamedTypeDependencyResolverPolicy.Resolve(IBuilderContext context) +101
   BuildUp_System.Web.Http.HttpConfiguration(IBuilderContext ) +202
   Microsoft.Practices.ObjectBuilder2.DynamicMethodBuildPlan.BuildUp(IBuilderContext context) +42
   Microsoft.Practices.ObjectBuilder2.BuildPlanStrategy.PreBuildUp(IBuilderContext context) +319
   Microsoft.Practices.ObjectBuilder2.StrategyChain.ExecuteBuildUp(IBuilderContext context) +260
   Microsoft.Practices.ObjectBuilder2.BuilderContext.NewBuildUp(NamedTypeBuildKey newBuildKey) +250
   Microsoft.Practices.Unity.ObjectBuilder.NamedTypeDependencyResolverPolicy.Resolve(IBuilderContext context) +101
   BuildUp_WebApi.Areas.HelpPage.Controllers.HelpController(IBuilderContext ) +204
```

Do you remember we added a test client to our project to test our API in my first article. That test client have a controller too, we need to override its settings to make our application work.Just go to Areas->HelpPage->Controllers->HelpController in WebAPI project like shown below,



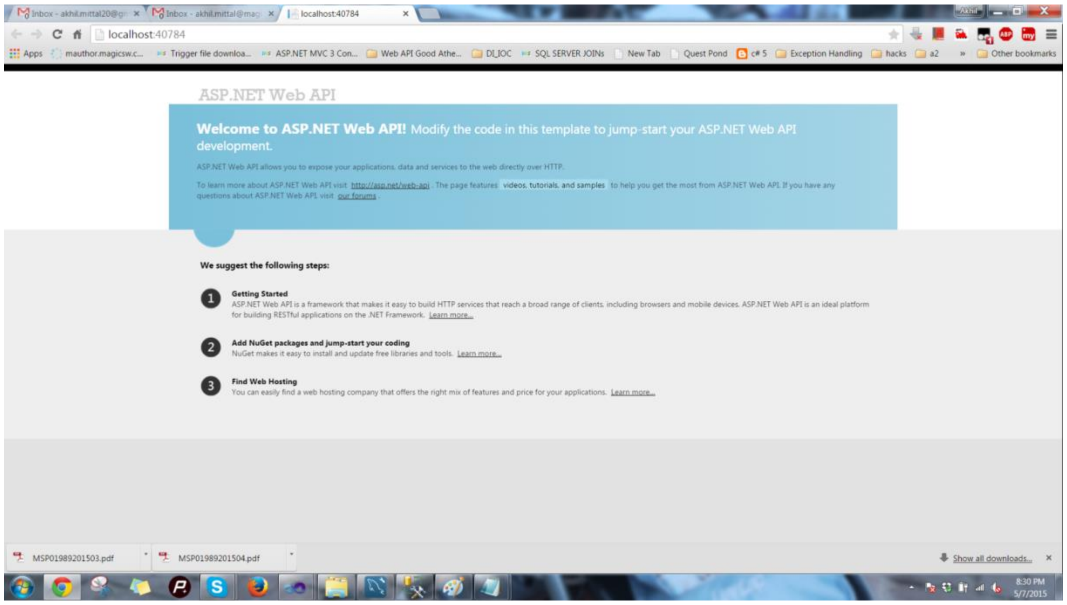
Comment out the existing constructors and add a Configuration property like shown below,

```
      //Remove constructors and existing Configuration property.

        //public HelpController()
        //    : this(GlobalConfiguration.Configuration)
        //{
        //}

        //public HelpController(HttpConfiguration config)
        //{
        //    Configuration = config;
        //}

        //public HttpConfiguration Configuration { get; private set; }

        /// <summary>
        /// Add new Configuration Property
        /// </summary>
        protected static HttpConfiguration Configuration
        {
            get { return GlobalConfiguration.Configuration; }
        }
```

Our controller code becomes,

```
using System;
using System.Web.Http;
using System.Web.Mvc;
using WebApi.Areas.HelpPage.Models;

namespace WebApi.Areas.HelpPage.Controllers
{
    /// <summary>
    /// The controller that will handle requests for the help page.
    /// </summary>
    public class HelpController : Controller
    {
        //Remove constructors and existing Configuration property.

        //public HelpController()
        //    : this(GlobalConfiguration.Configuration)
        //{
        //}

        //public HelpController(HttpConfiguration config)
        //{
        //    Configuration = config;
        //}

        //public HttpConfiguration Configuration { get; private set; }

        /// <summary>
        /// Add new Configuration Property
        /// </summary>
        protected static HttpConfiguration Configuration
        {
            get { return GlobalConfiguration.Configuration; }
        }

        public ActionResult Index()
        {
            return View(Configuration.Services.GetApiExplorer().ApiDescriptions);
        }

        public ActionResult Api(string apiId)
        {
            if (!String.IsNullOrEmpty(apiId))
            {
                HelpPageApiModel apiModel = Configuration.GetHelpPageApiModel(apiId);
                if (apiModel != null)
                {
                    return View(apiModel);
                }
            }

            return View("Error");
        }
    }
}
```

Just run the application, we get,


ASP.NET Web API

Welcome to ASP.NET Web API! Modify the code in this template to jump-start your ASP.NET Web API development.

ASP.NET Web API allows you to expose your applications, data and services to the web directly over HTTP.

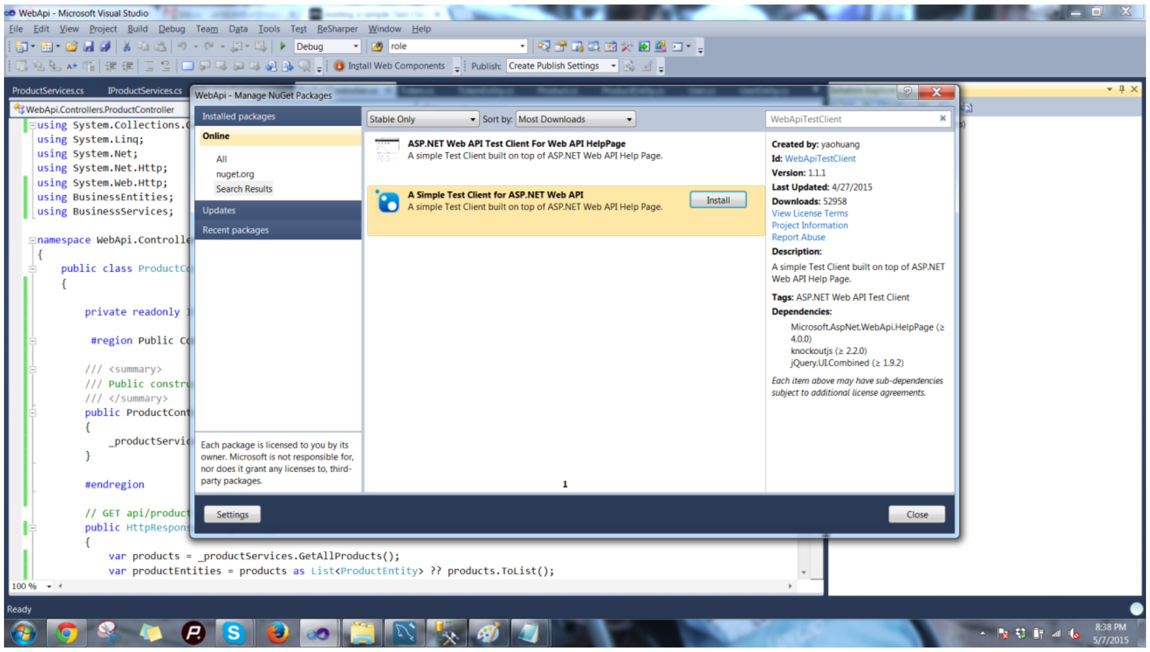
To learn more about ASP.NET Web API visit http://asp.net/web-api . The page features videos, tutorials, and samples to help you get the most from ASP.NET Web API. If you have any questions about ASP.NET Web API, visit our forums .

We suggest the following steps:

1. **Getting Started**
   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. Learn more...
2. **Add NuGet packages and jump-start your coding**
   NuGet makes it easy to install and update free libraries and tools. Learn more...
3. **Find Web Hosting**
   You can easily find a web hosting company that offers the right mix of features and price for your applications. Learn more...

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

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


WebApi - Manage NuGet Packages

Stable Only | Sort by: Most Downloads | WebApiTestClient

ASP.NET Web API Test Client For Web API HelpPage
A simple Test Client built on top of ASP.NET Web API Help Page.

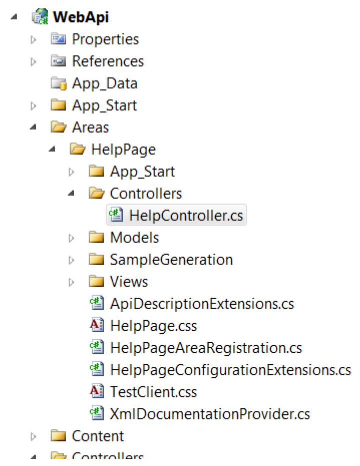
A Simple Test Client for ASP.NET Web API
A simple Test Client built on top of ASP.NET Web API Help Page.

Created by: yaohuang
Id: WebApiTestClient
Version: 1.1.1
Last Updated: 4/27/2015
Downloads: 52958
View License Terms
Project Information
Report Abuse
Description:
A simple Test Client built on top of ASP.NET Web API Help Page.
Tags: ASP.NET Web API Test Client
Dependencies:
Microsoft.AspNet.WebApi.HelpPage (≥ 4.0.0)
knockoutjs (≥ 2.2.0)
jQuery.UI.Combined (≥ 1.9.2)
Each item above may have sub-dependencies subject to additional license agreements.

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

```
using System.Collections.
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using BusinessEntities;
using BusinessServices;

namespace WebApi.Controll
{
    public class ProductC
    {

        private readonly I

         #region Public C

        /// <summary>
        /// Public constru
        /// </summary>
        public ProductCont
        {
            _productServi
        }

        #endregion

        // GET api/product
        public HttpRespons
        {
            var products = _productServices.GetAllProducts();
            var productEntities = products as List<ProductEntity> ?? products.ToList();
```

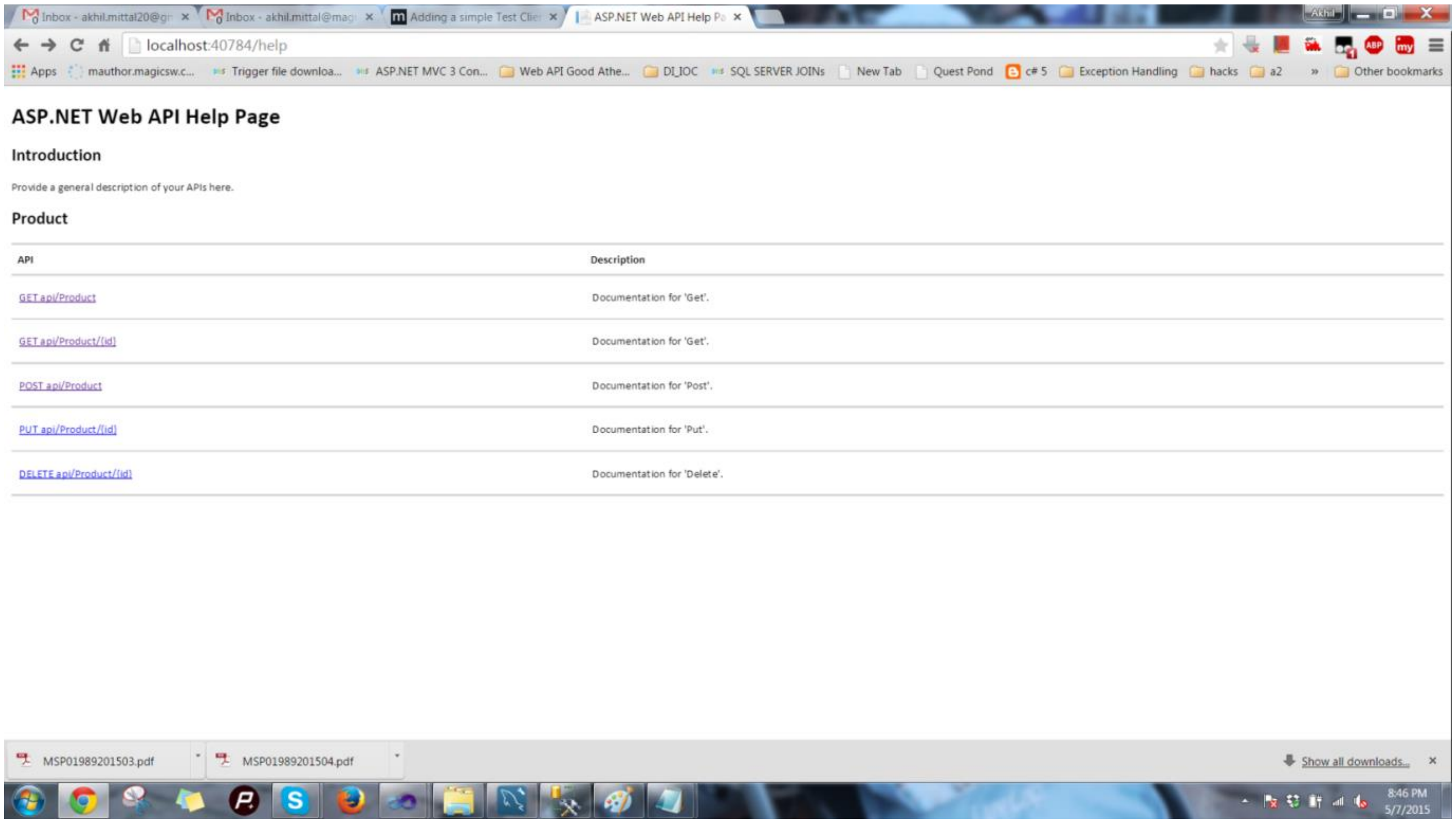
You'll get "A simple Test Client for ASP.NET Web API", just add it. You'll get a help controller in Areas-> HelpPage like shown below,



```
WebApi
  Properties
  References
  App_Data
  App_Start
  Areas
    HelpPage
      App_Start
      Controllers
        HelpController.cs
      Models
      SampleGeneration
      Views
      ApiDescriptionExtensions.cs
      HelpPage.css
      HelpPageAreaRegistration.cs
      HelpPageConfigurationExtensions.cs
      TestClient.css
      XmlDocumentationProvider.cs
  Content
```

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,



ASP.NET Web API Help Page

Introduction

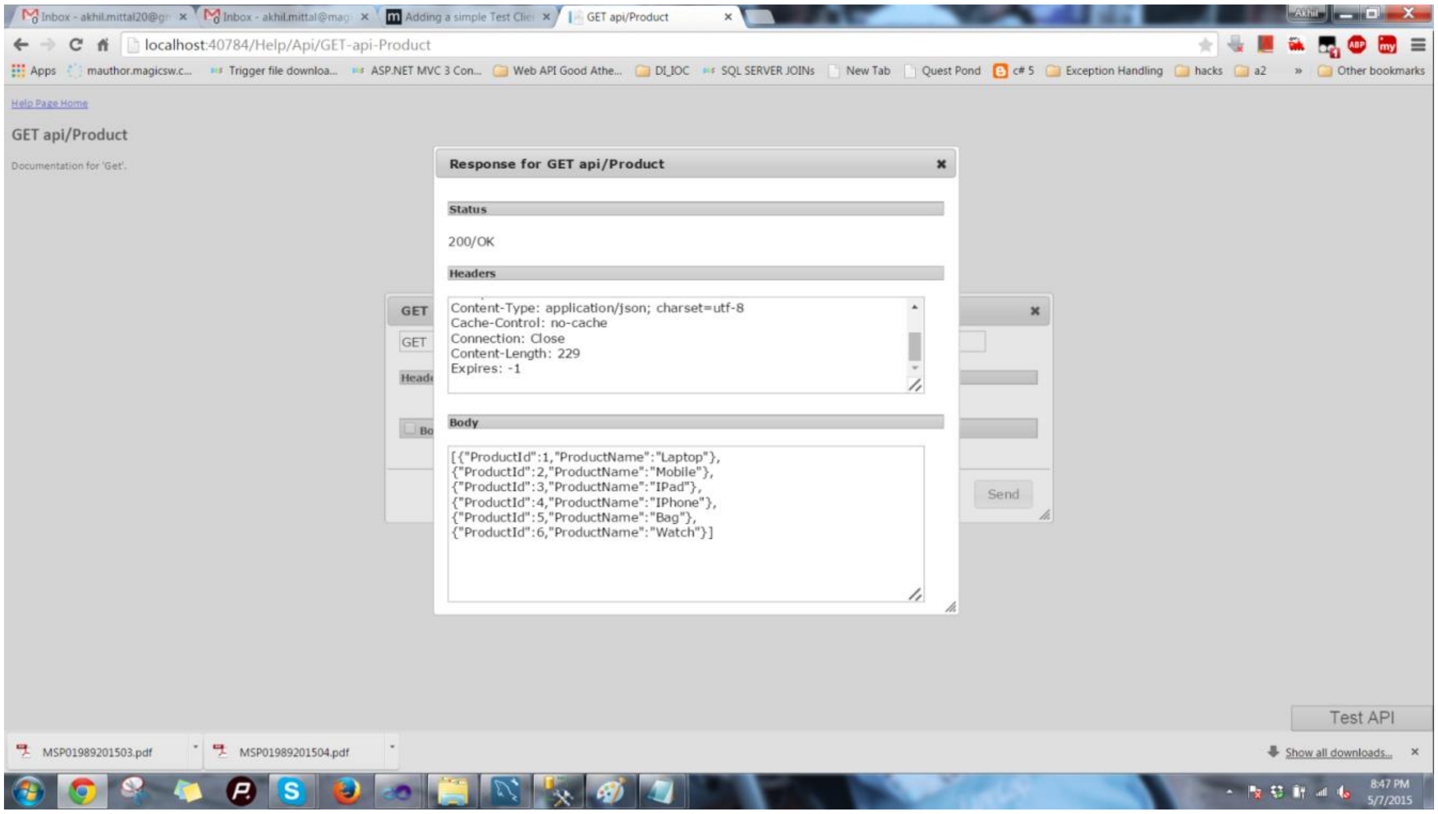
Provide a general description of your APIs here.

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'.

You can test each service by clicking on it.

Service for GetAllProduct,

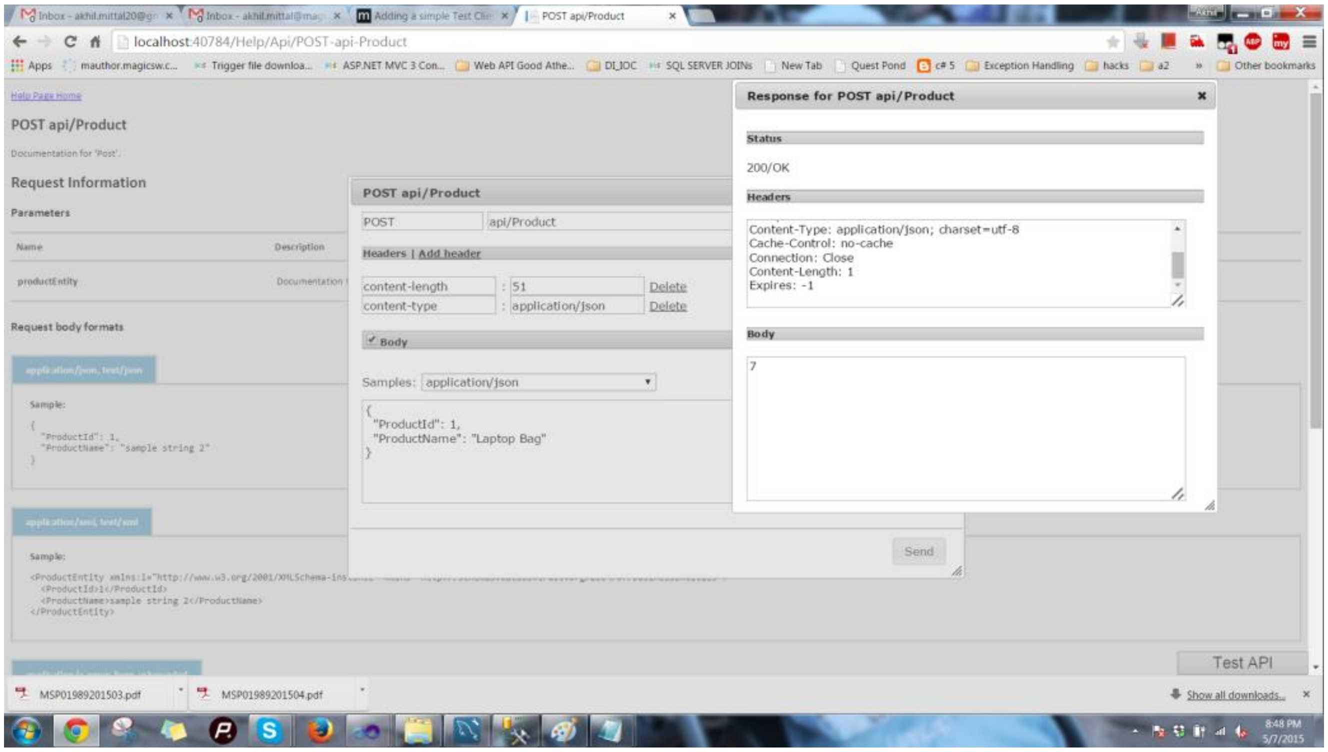

localhost:40784/Help/Api/GET-api-Product
GET api/Product
Documentation for 'Get'.
Response for GET api/Product
Status
Headers
Body
Send
Test API

200/OK

```
Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 229
Expires: -1
```

```
[{"ProductId":1,"ProductName":"Laptop"},
{"ProductId":2,"ProductName":"Mobile"},
{"ProductId":3,"ProductName":"IPad"},
{"ProductId":4,"ProductName":"IPhone"},
{"ProductId":5,"ProductName":"Bag"},
{"ProductId":6,"ProductName":"Watch"}]
```

For Create a new product,



content-length	51	Delete
content-type	application/json	Delete

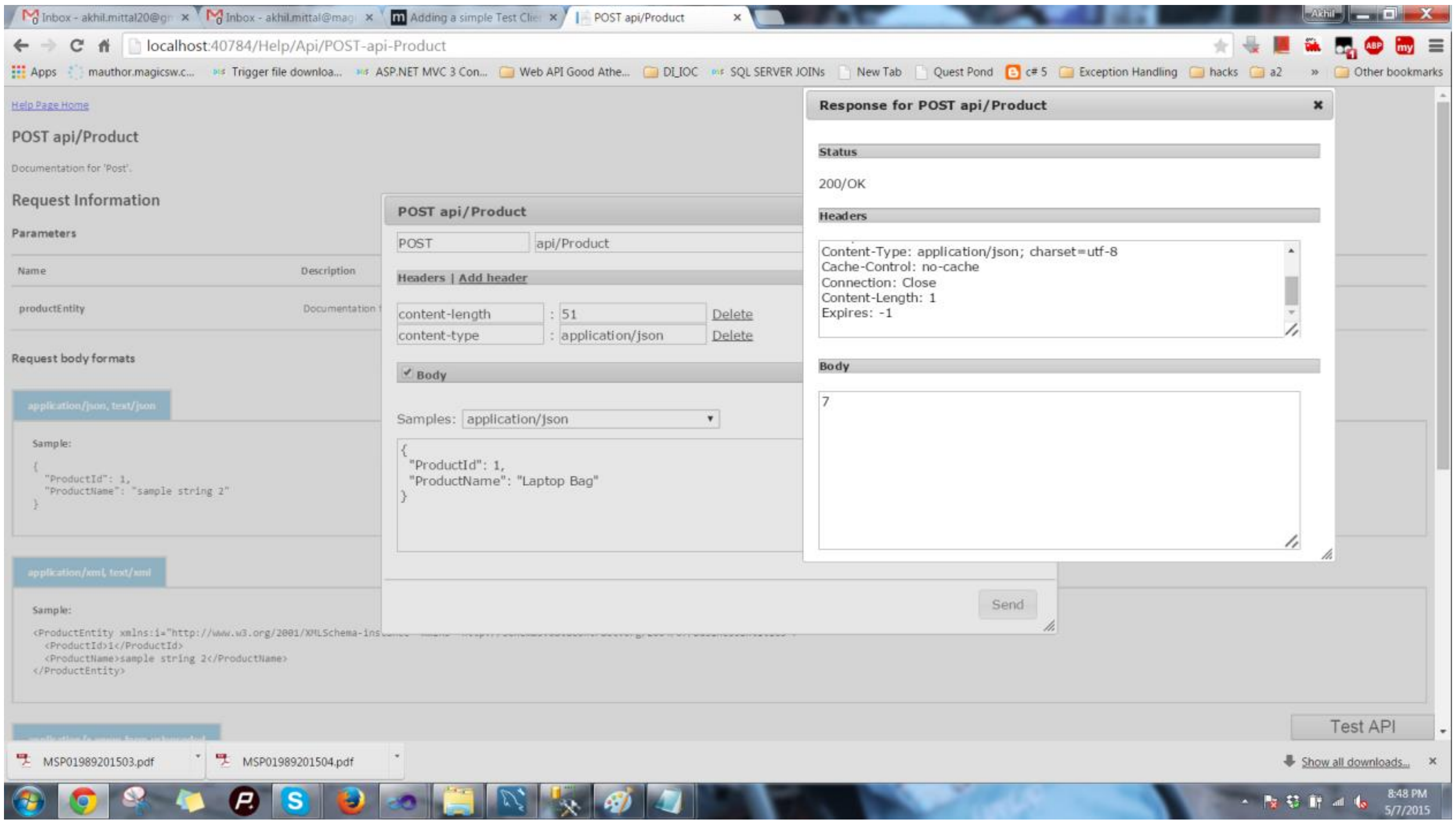
```
{
  "ProductId": 1,
  "ProductName": "Laptop Bag"
}
```

200/OK

```
Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 1
Expires: -1
```

```
7
```

In database, we get new product,



Help Page Home

POST api/Product

Documentation for 'Post'.

Request Information

Parameters

Name	Description
productEntity	Documentation

Request body formats

application/json, text/json

Sample:

```
{
  "ProductId": 1,
  "ProductName": "sample string 2"
}
```

application/xml, text/xml

Sample:

```
<ProductEntity xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/BusinessEntities">
  <ProductId>1</ProductId>
  <ProductName>sample string 2</ProductName>
</ProductEntity>
```

POST api/Product

POST api/Product

Headers | Add header

content-length	51	Delete
content-type	application/json	Delete

Body

Samples: application/json

```
{
  "ProductId": 1,
  "ProductName": "Laptop Bag"
}
```

Send

Response for POST api/Product

Status

200/OK

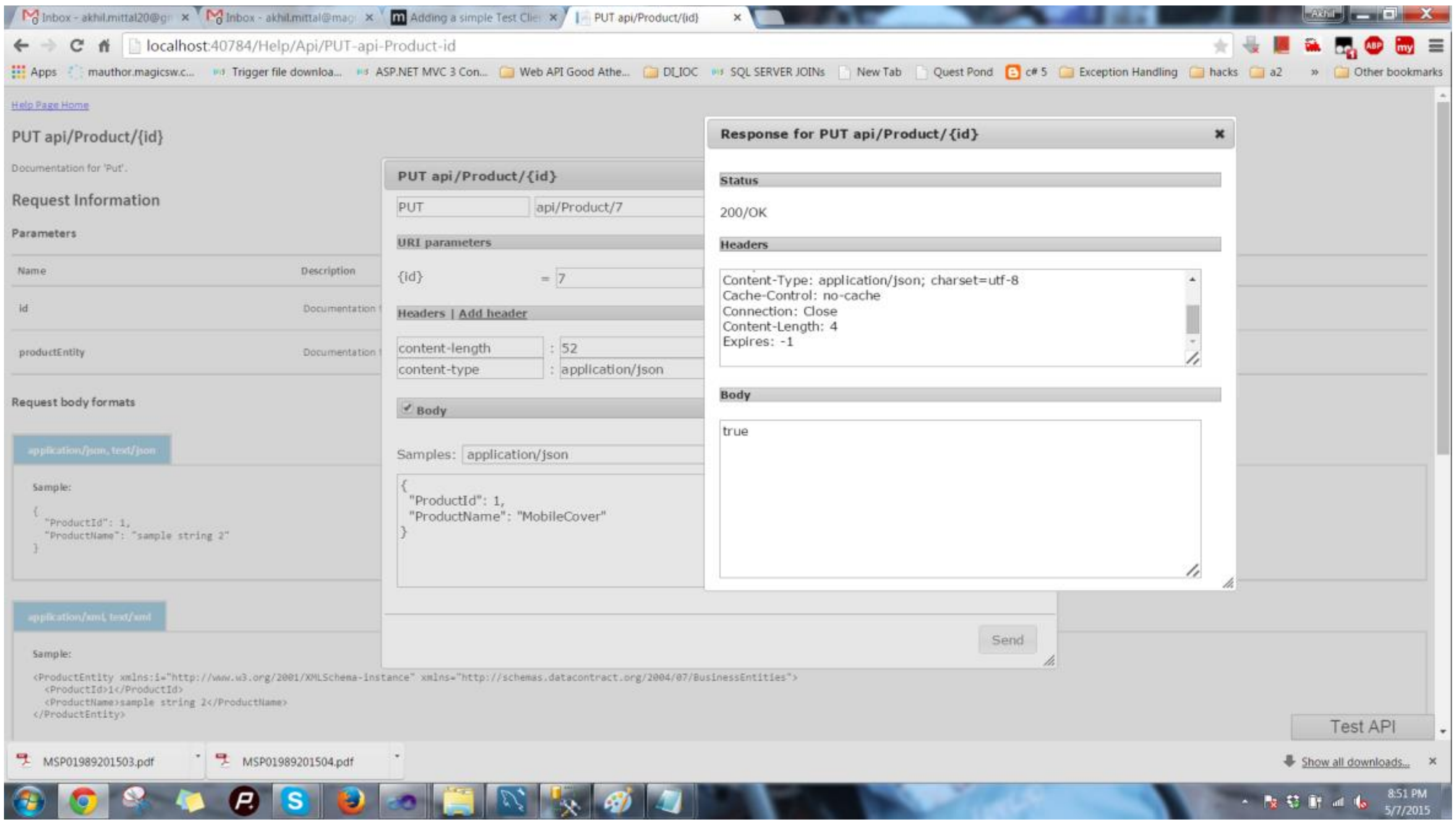
Headers

```
Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 1
Expires: -1
```

Body

```
7
```

Update product:



Help Page Home

PUT api/Product/{id}

Documentation for 'Put'.

Request Information

Parameters

Name	Description
id	Documentation
productEntity	Documentation

Request body formats

application/json, text/json

Sample:

```
{
  "ProductId": 1,
  "ProductName": "sample string 2"
}
```

application/xml, text/xml

Sample:

```
<ProductEntity xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/BusinessEntities">
  <ProductId>1</ProductId>
  <ProductName>sample string 2</ProductName>
</ProductEntity>
```

PUT api/Product/{id}

PUT api/Product/7

URI parameters

{id} = 7

Headers | Add header

content-length	52
content-type	application/json

Body

Samples: application/json

```
{
  "ProductId": 1,
  "ProductName": "MobileCover"
}
```

Send

Response for PUT api/Product/{id}

Status

200/OK

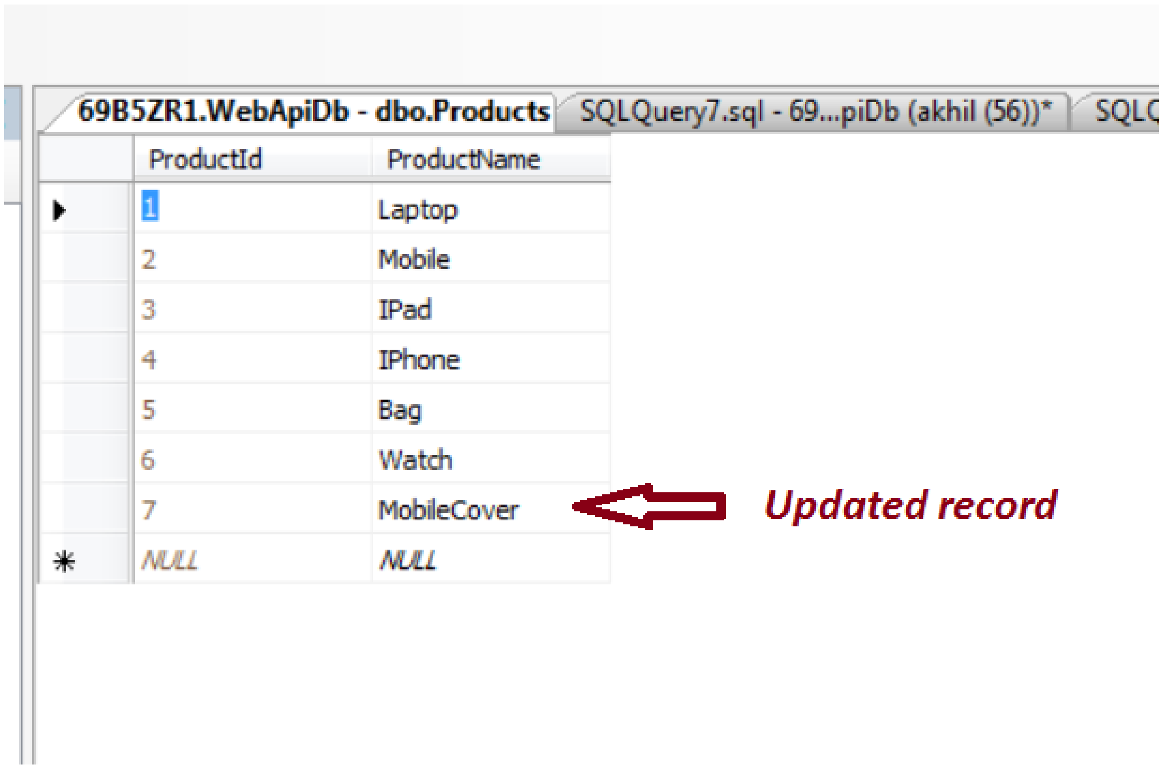
Headers

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

Body

```
true
```

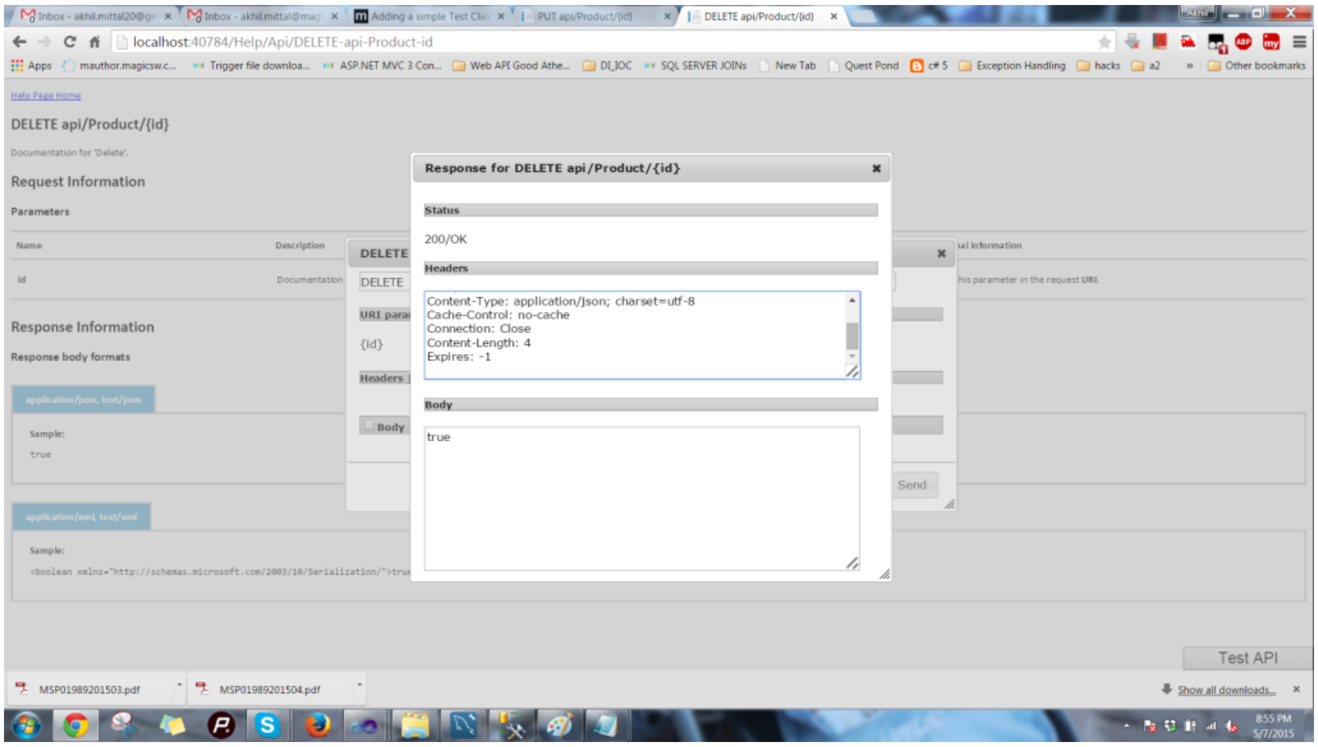
We get in database,



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

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

Delete product:



localhost:40784/Help/Api/DELETE-api-Product-id

DELETE api/Product/{id}

Documentation for 'Delete'.

Request Information

Parameters

Response Information

Response body formats

Sample:
true

Sample:
<boolean xmlns="http://schemas.microsoft.com/2003/10/Serialization/">true

Response for DELETE api/Product/{id}

Status

200/OK

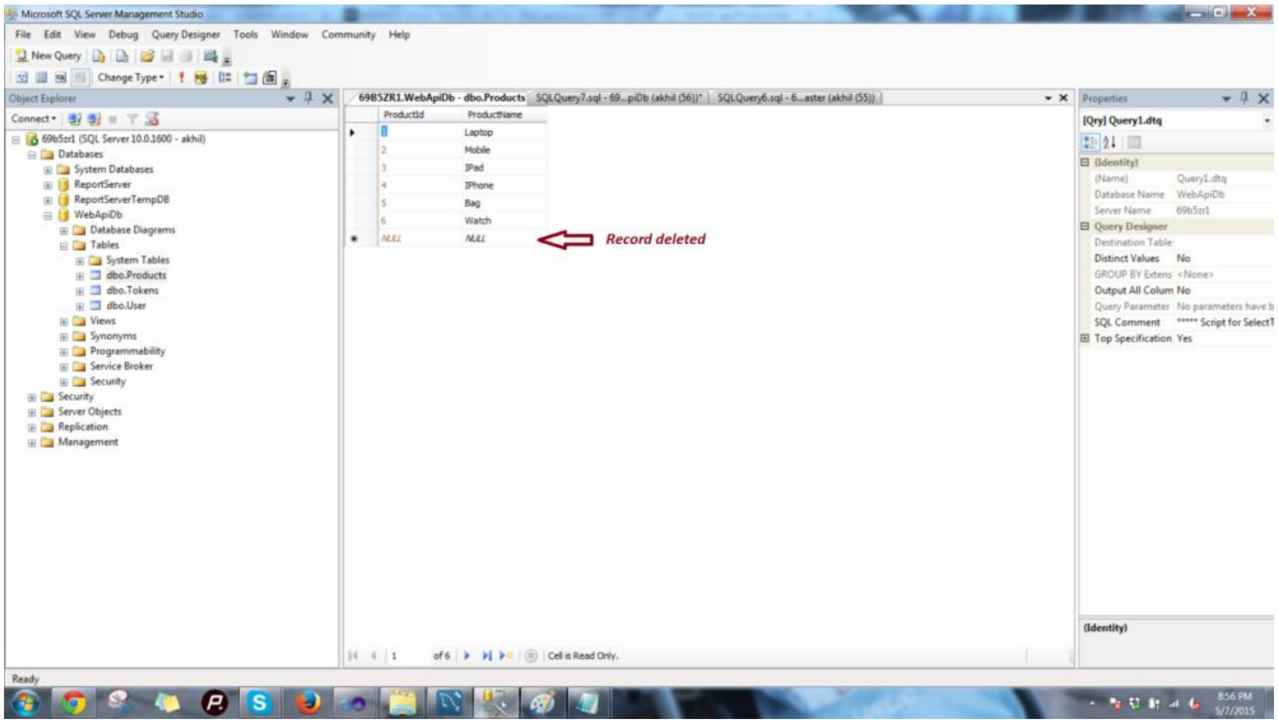
Headers

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

Body

```
true
```

In database:


Microsoft SQL Server Management Studio
69B5ZR1.WebApiDb - dbo.Products
Record deleted
Cell is Read Only.

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

Job done.

Image source: http://codeopinion.com/wp-content/uploads/2015/02/injection.jpg

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 while writing our first application?
Our API talks to Services and Services talk to DataModel. We'll never allow DataModel talk to APIs for security reasons. But did you notice that when we were registering the type in Bootstrapper class, we also registered the type of UnitOfWork that means we added DataModel as a reference to our API project. This is a design breach. We tried to resolve dependency of a dependency by violating our design and compromising security.
In my next article, we'll overcome this situation, we'll try to resolve dependency and its dependency without violating our design and compromising security.Infact we'll make it more secure and loosely coupled.
In my next article we'll make use of [Managed Extensibility Framework(MEF)]() to achieve the same.

Conclusion

We now know how to use Unity container to resolve dependency and perform inversion of control.
But still there are some flaws in this design. In my next article I'll try to make system more strong. 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.

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).