RESTful Day #3: Resolve dependency of dependencies using Inversion of Control and dependency injection in Asp.net Web APIs with Unity Container and Managed Extensibility Framework (MEF)

–by Akhil Mittal

Table of Contents

Introduction

In my last two articles I explained how to create a RESTful service using Asp.Net Web API working with Entity Framework and resolving dependencies using Unity Container. In this article I'll explain how to create a loosely coupled system with Unity Container and MEF(Managed Extensibility Framework) using Inversion of Control. I'll not be explaining much theory but rather focus more on practical implementations. For the readers who are following this series, they can use their existing solution that they have created till time. For my new readers of this article, I have provided the download link for the previous source code and current source code as well.

For theory and understanding of DI and IOC you can follow the following links: Unity and Inversion of Control(IOC).

Roadmap

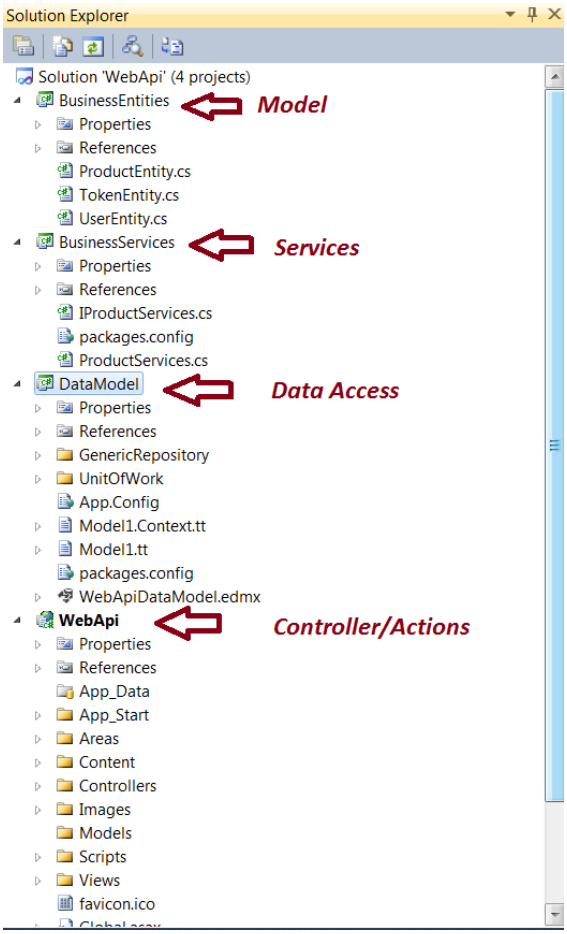
Here is my roadmap for learning RESTful APIs,

- 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: Resolve dependency of dependencies using Inversion of Control and dependency injection in Asp.net Web APIs with Unity Container and Managed Extensibility Framework (MEF).**
- RESTful Day #4: Custom URL Re-Writing/Routing using Attribute Routes in MVC 4 Web APIs.
- RESTful Day #5: 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

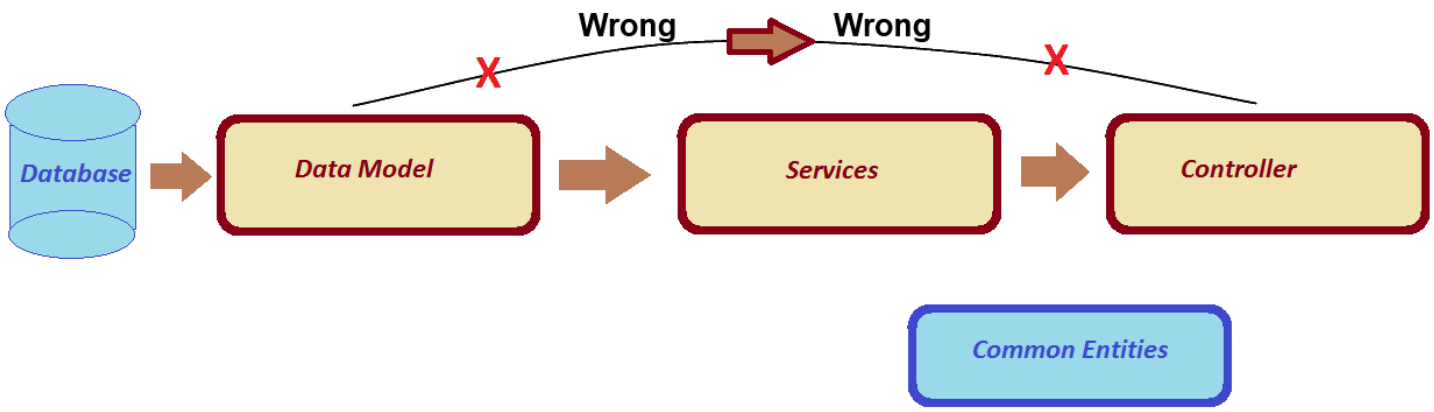
We already have an existing design. If you open the solution, you'll get to see the structure as mentioned below,



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 article 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.

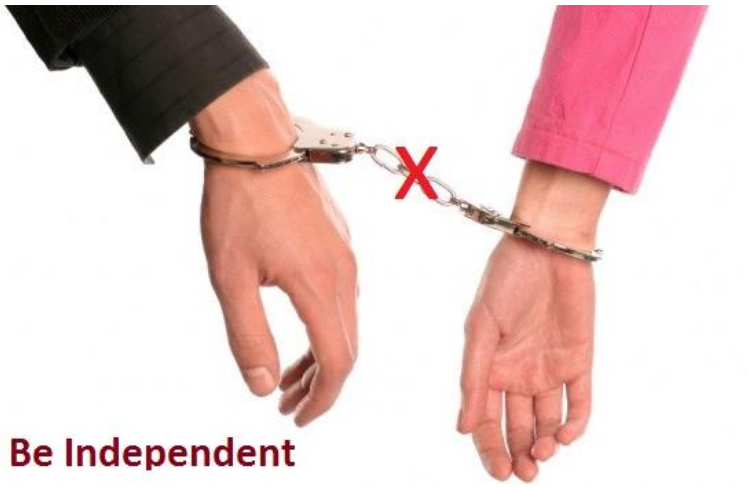


Image source : http://o5.com/wp-content/uploads/2010/08/dreamstime_15583711-550x371.jpg

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)

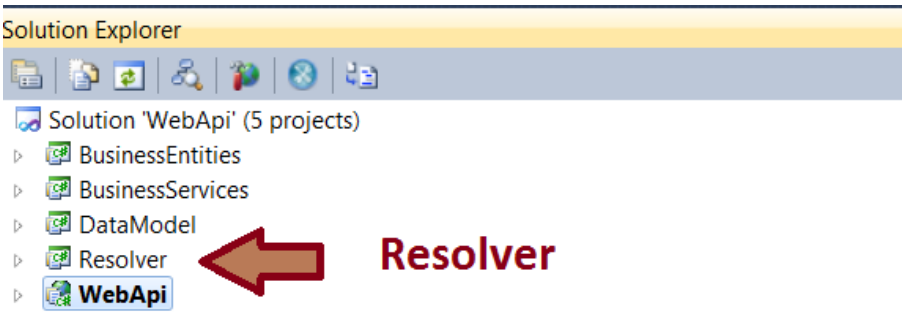
You can have a read about Unity from msdn link. I am just quoting some lines from msdn,

> *"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

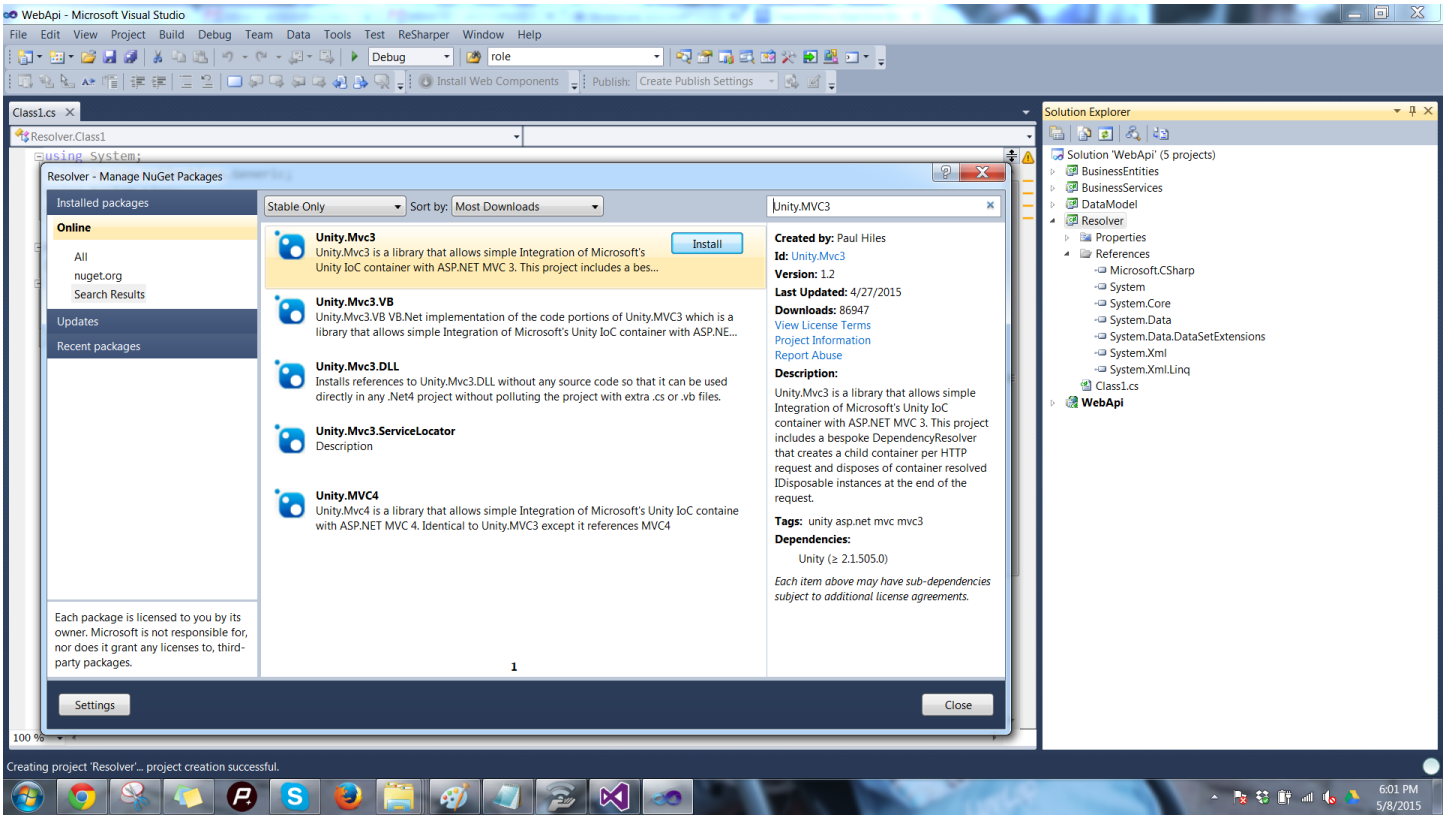
Open your Visual studio, I am using VS 2010, you can use VS version 2010 or above. Load the solution.

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



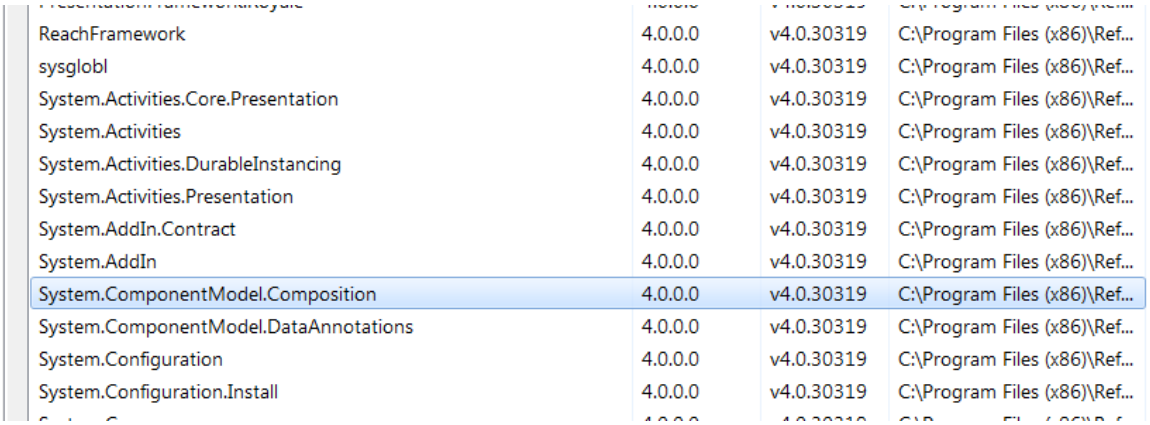
I have intentionally chosen this name, and you already know it why ☺

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);
    }
}
```

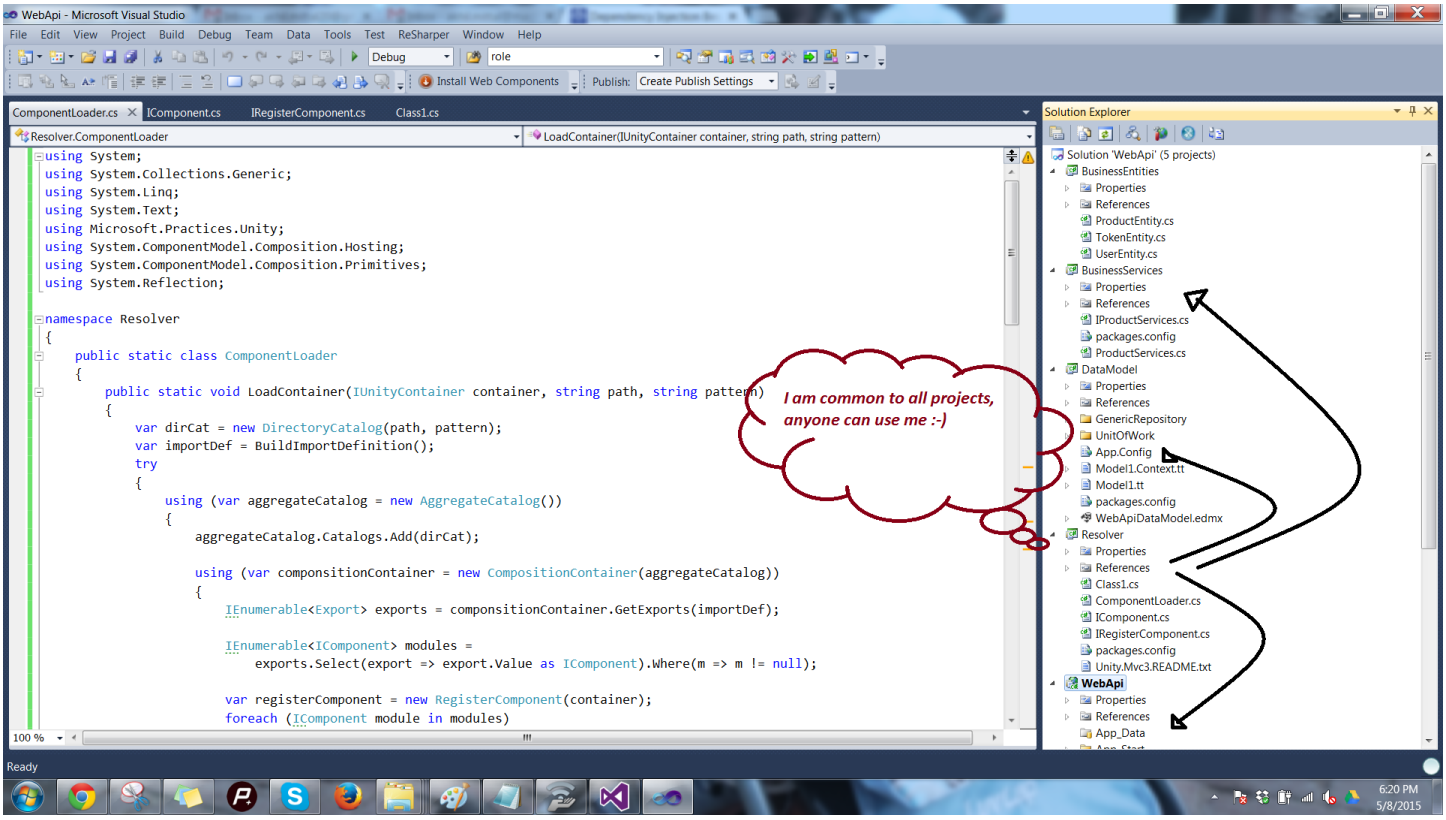
Step 6: Now we'll write a packager or you can say a wrapper over MEF and Unity to register types/ components. This is the core MEF implementation. Create a class named ComponentLoader, and add following code to it,

```
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 componsitionContainer = new
CompositionContainer(aggregateCatalog))
                    {
                        IEnumerable<Export> exports =
componsitionContainer.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,

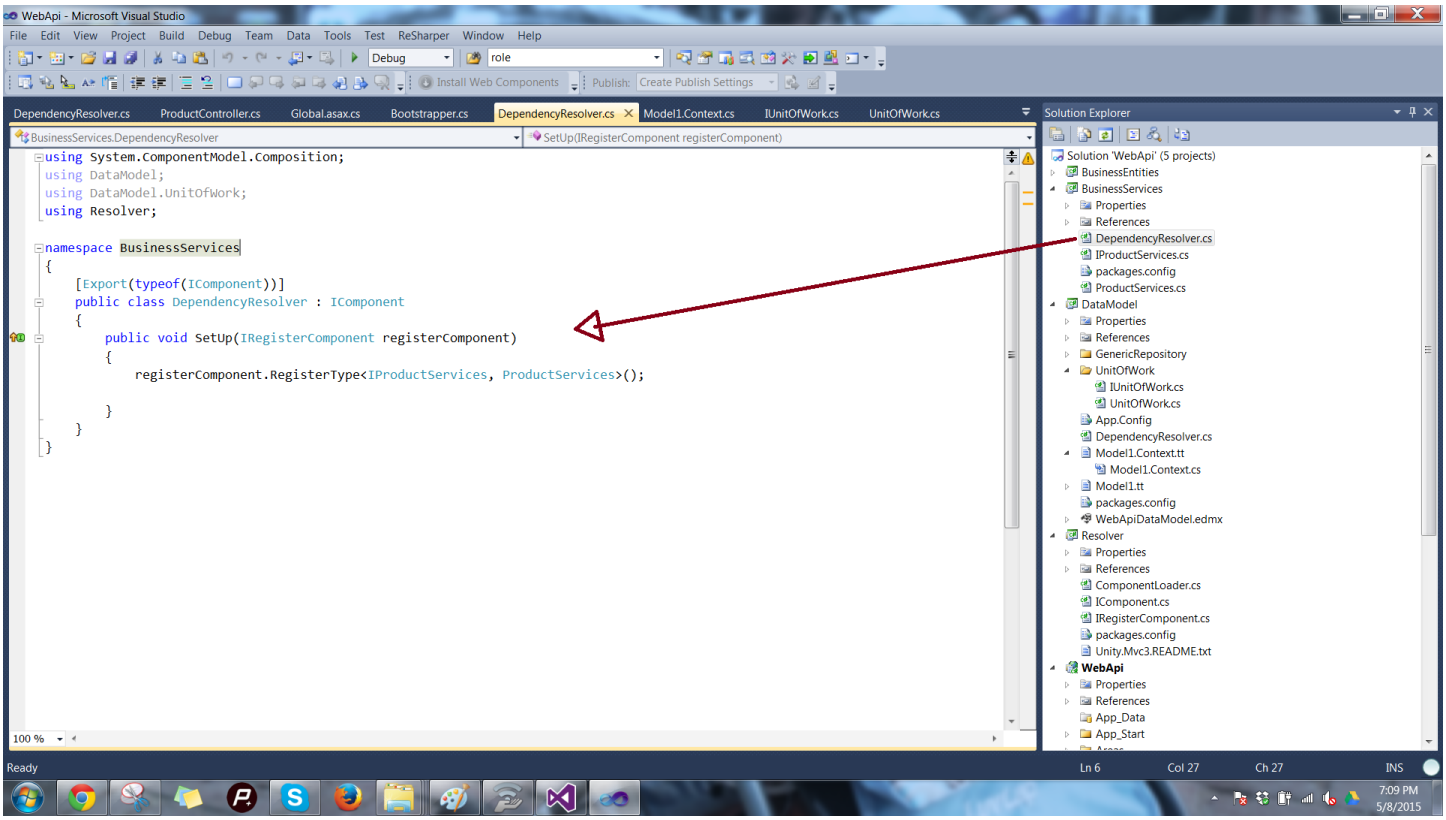


Setup Business Services

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

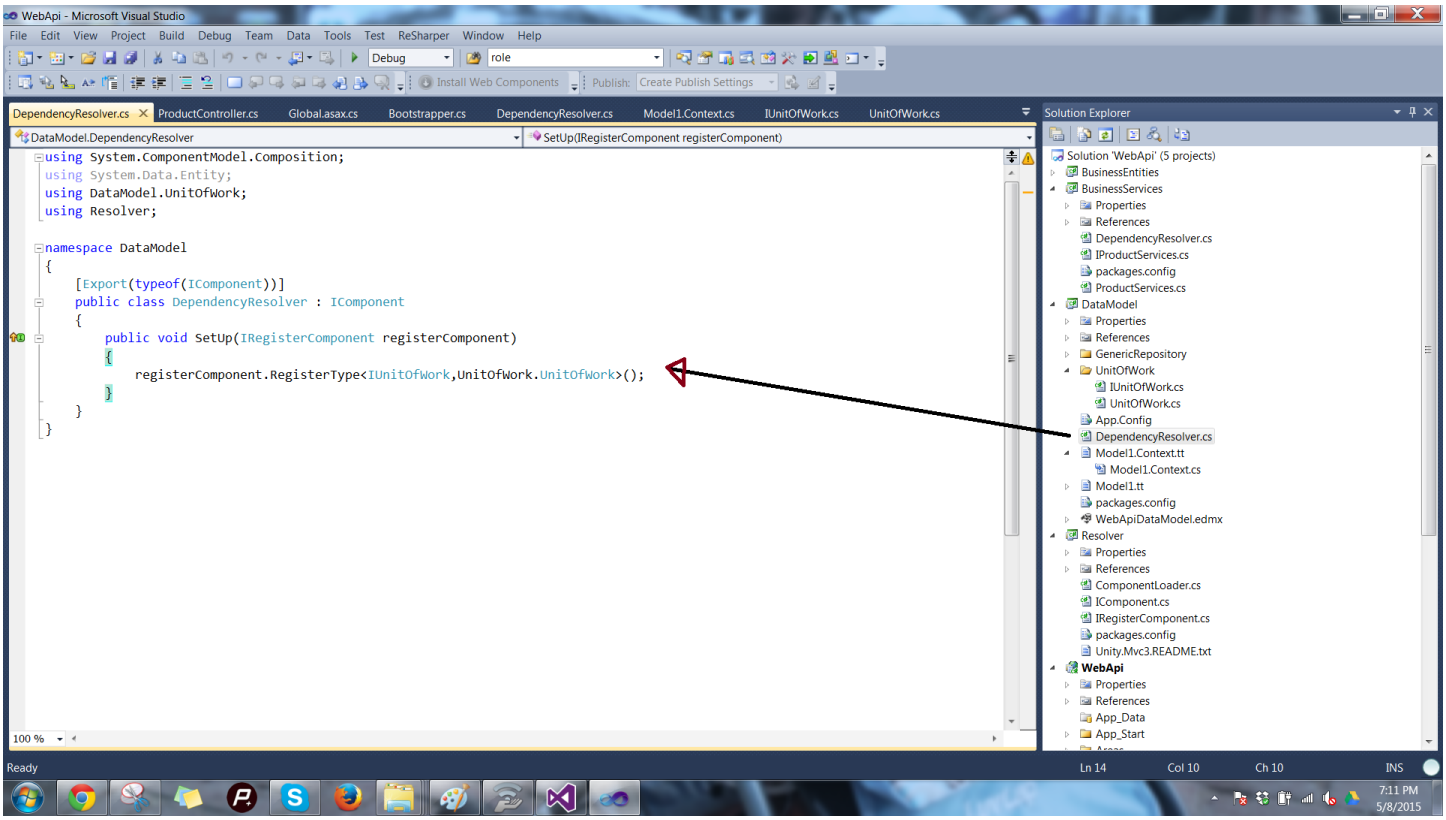
We have added Resolver project reference to DataModel project as well. So we'll try to register the type of UnitOfWork in this project.We proceed in same fashion, just add a DependencyResolver class and implement its Setup method to register type of UnitOfWork.To make the code more readable and standard, 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 DepenencyResolver 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;
        }
    }
}
```

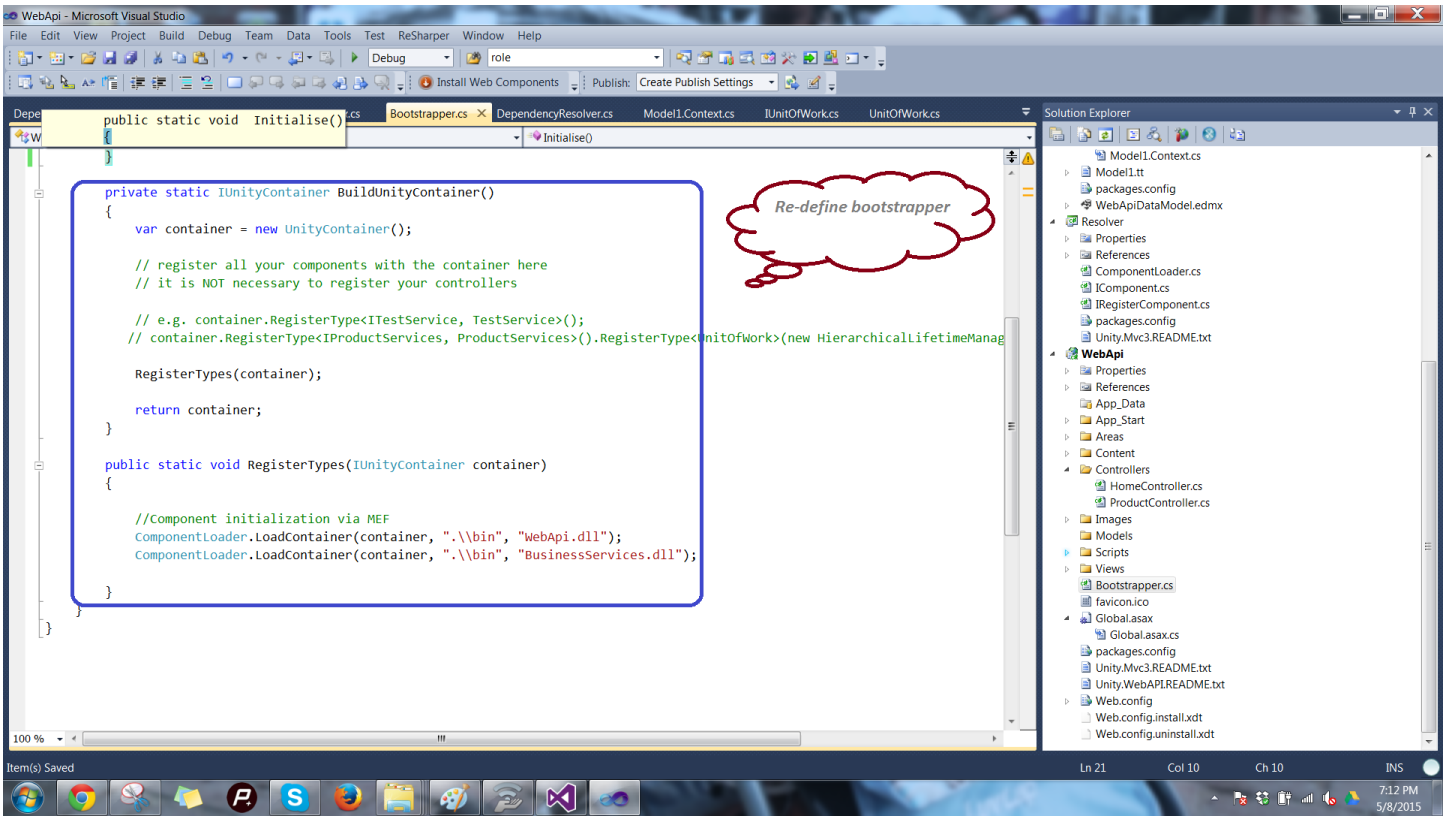
Now, we need to change the code base a bit to make our system loosely coupled.Just remove the reference of DataModel from WebAPI project.

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.




```
public static void  Initialise()
{
}

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 HierarchicalLifetimeManag

    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");

}
}
}
```

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.

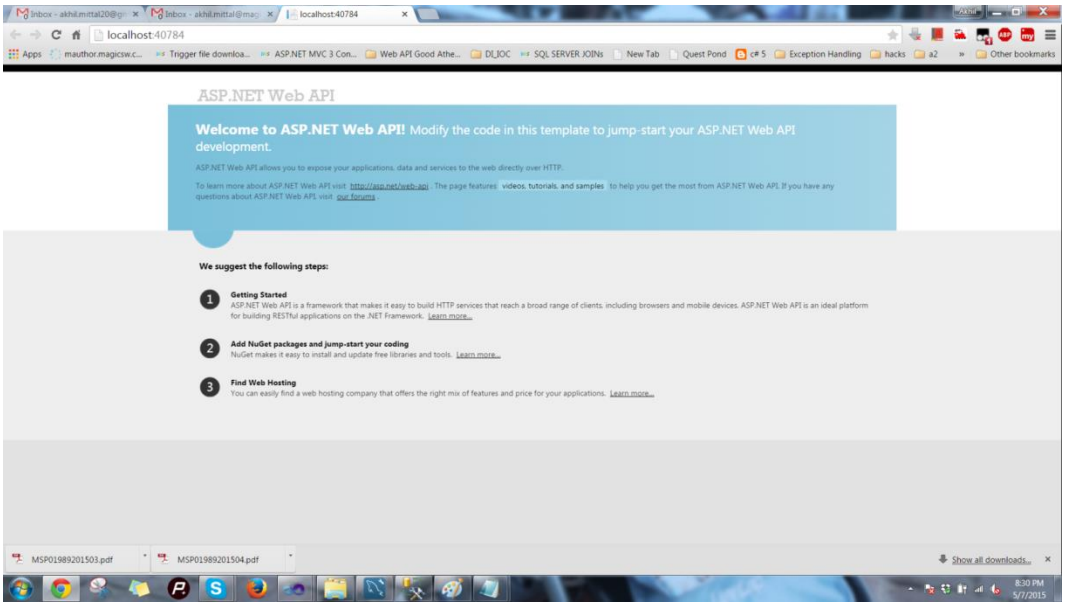
Had the name of BusinessServicess.dll be WebAPI.Services.dll, then we would have only written one line of code to load both the WebAPI and BusinessService dll like shown below,

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

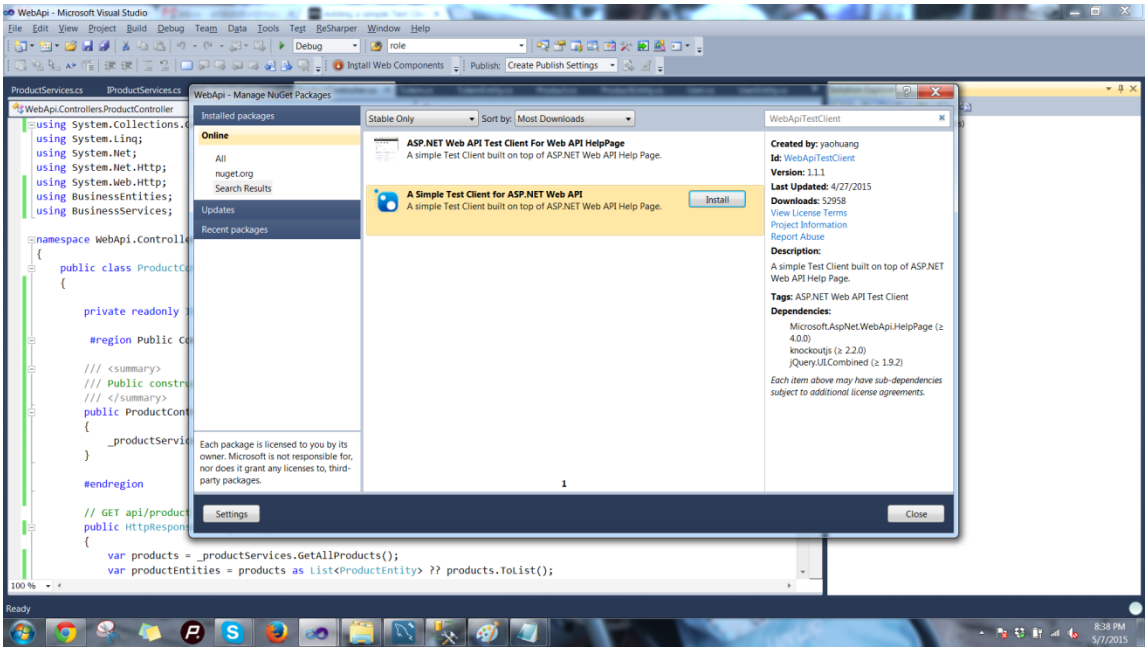
Yes we can make use of Regex.

Running the application

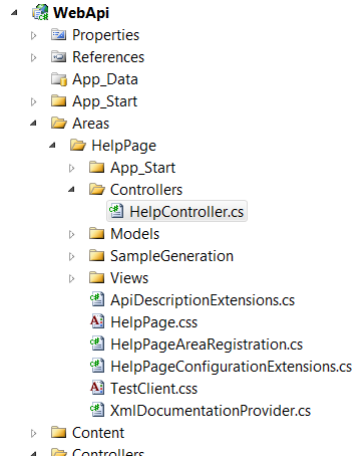
Just run the application, we get,



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,

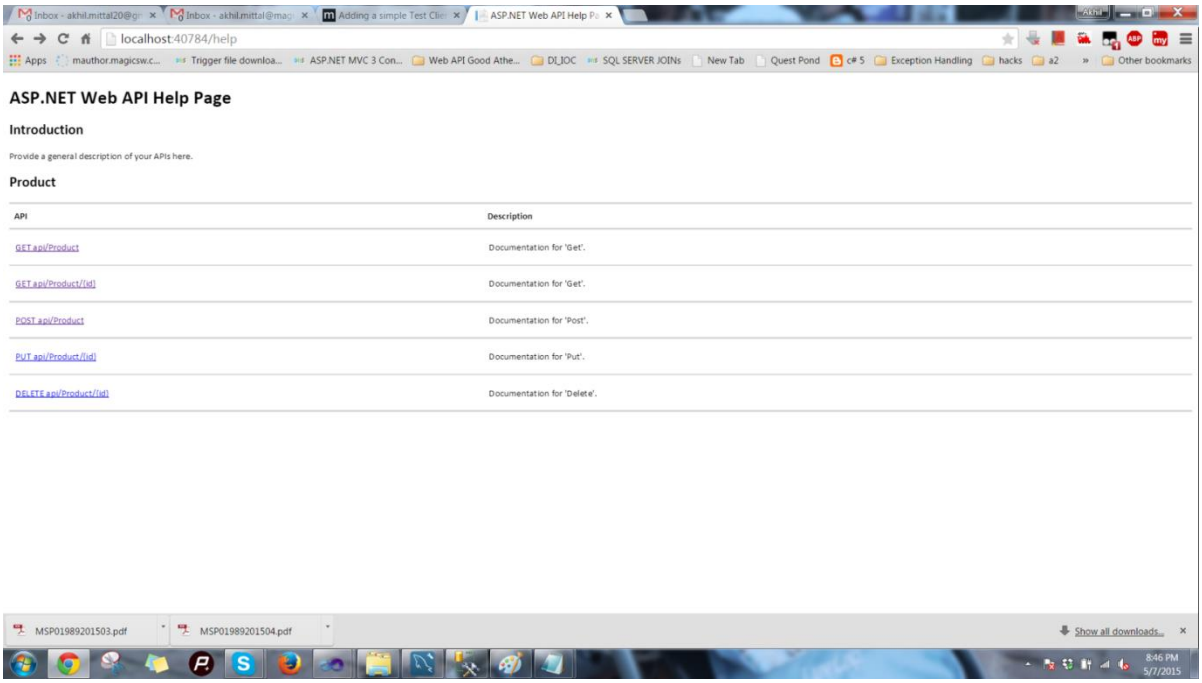


You'll get "A simple Test Client for ASP.NET Web API", just add it. You'll get a help controller in Areas-> HelpPage like shown below,

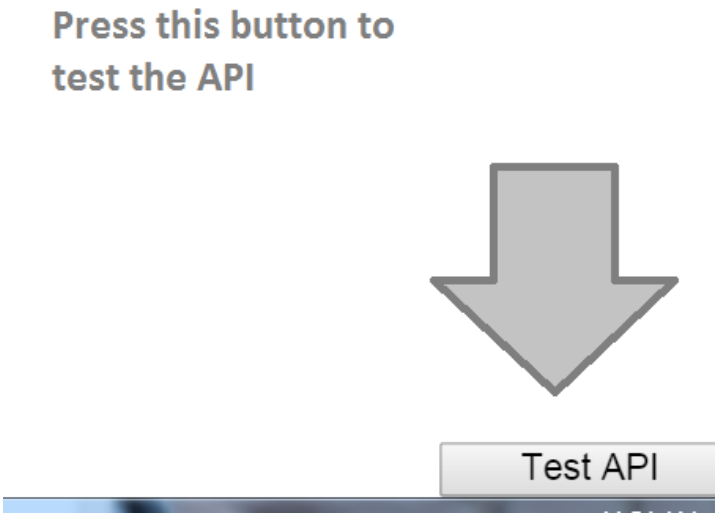


I have already provided the database scripts and data in my previous article, you can use the same.

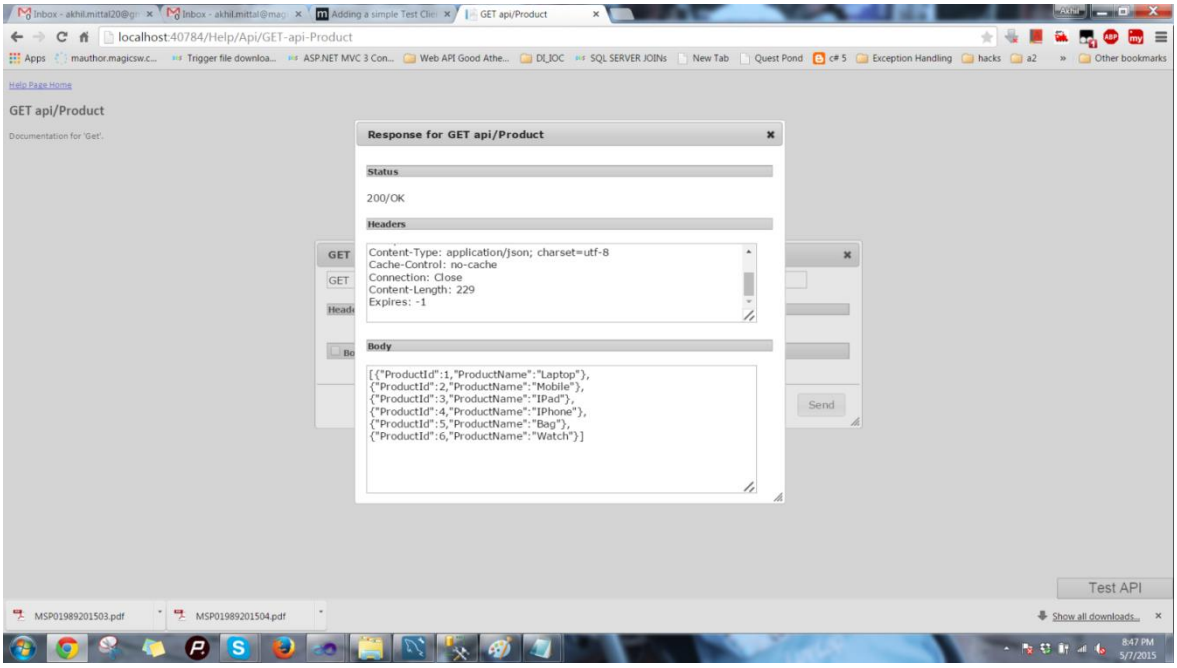
Append “/help” in the application url, and you’ll get the test client,



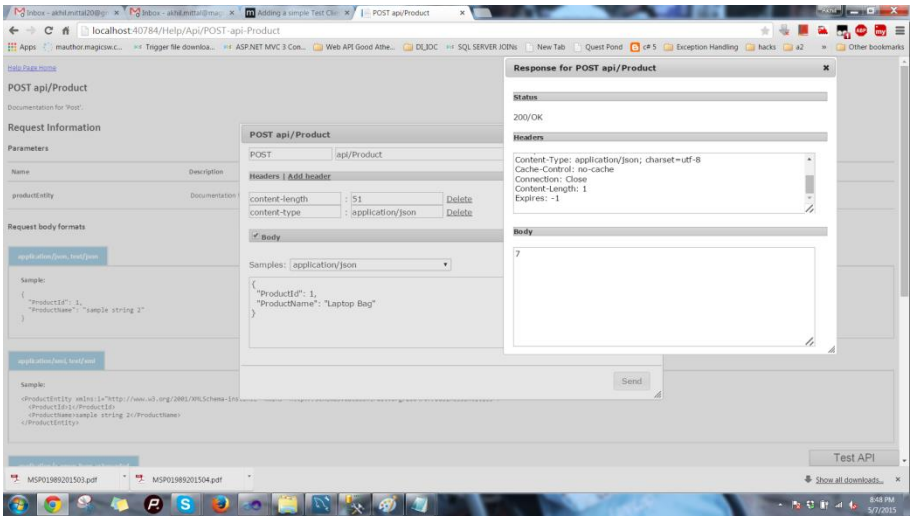
You can test each service by clicking on it. Once you click on the service link, you'll be redirected to test the service page of that particular service.On that page there is a button Test API in the right bottom corner, just press that button to test your service,



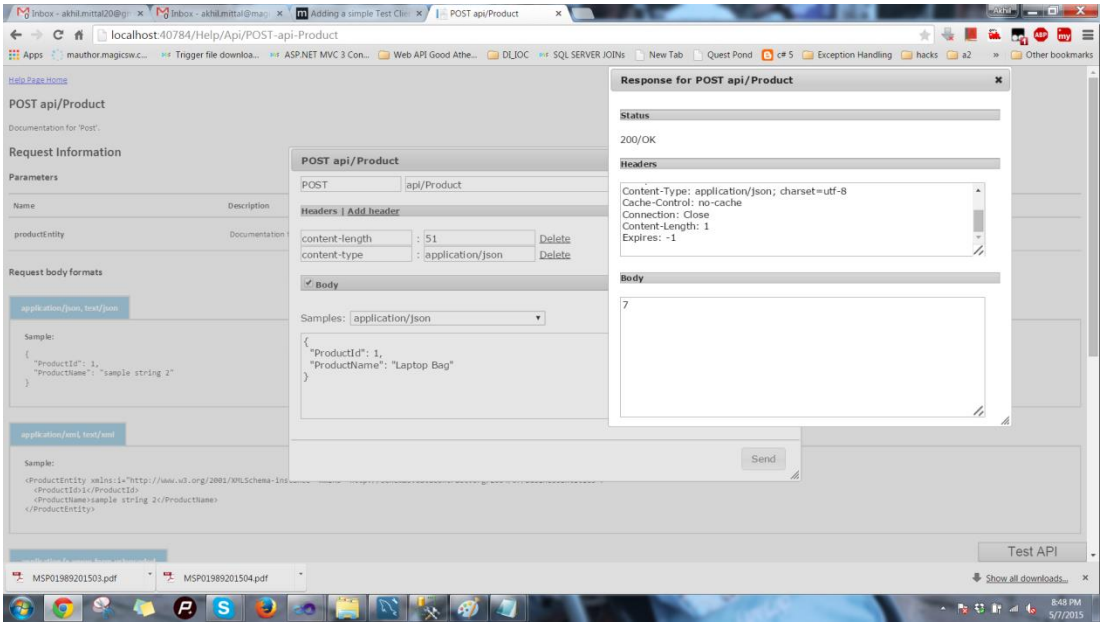
Service for GetAllProduct,



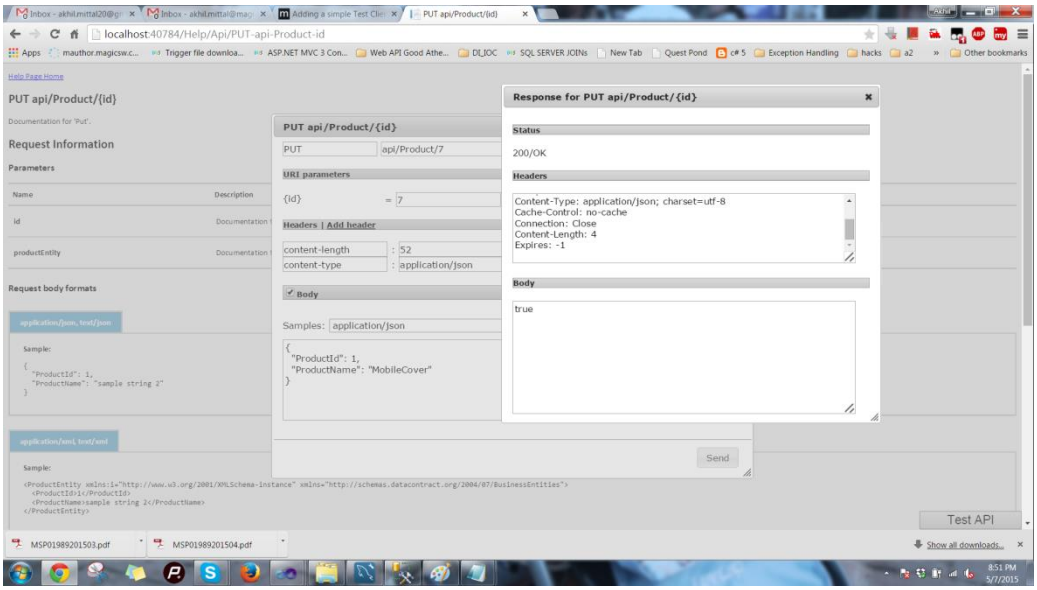
For Create a new product,



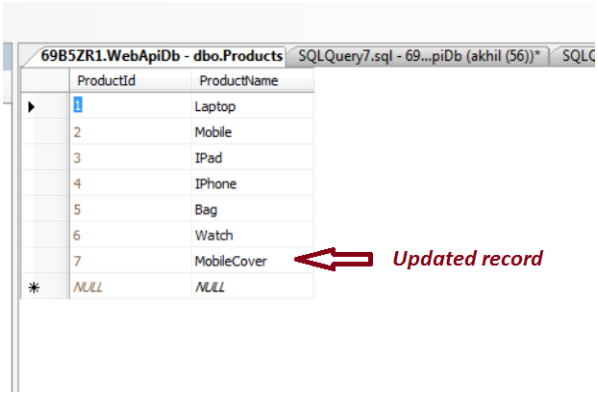
In database, we get new product,



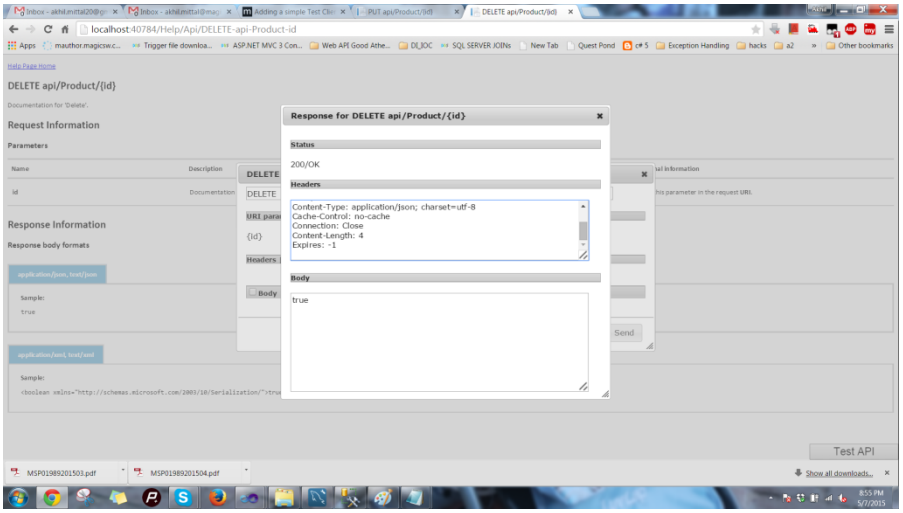
Update product:



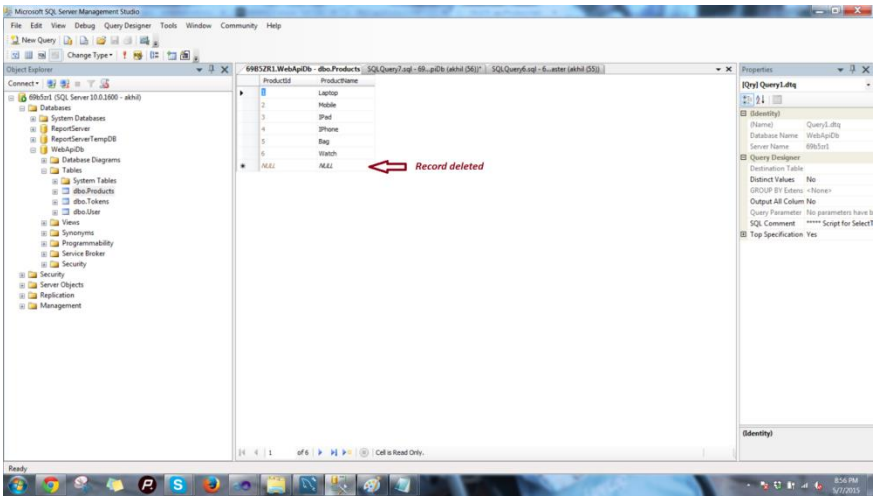
We get in database,



Delete product:



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 my next article 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.

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).