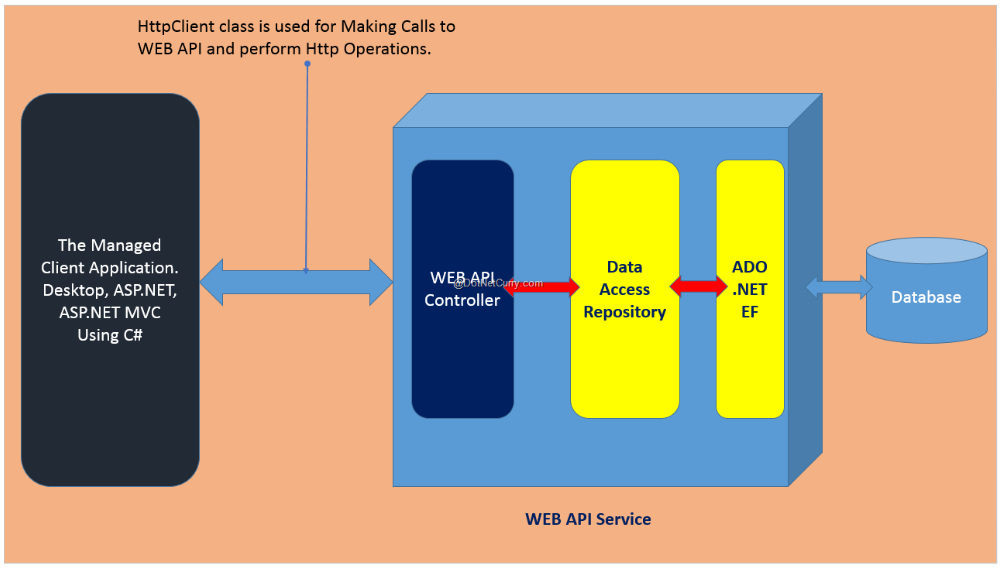
Consuming a Web API Asynchronously in ASP.NET MVC or WPF: http://www.dotnetcurry.com/aspnet/1192/aspnet-web-api-async-calls-mvc-wpf

MVC_ClientWebAPI & APIService: Creates both the service and the MVC Client for consumption (*Painful and very complete*)

One of the benefits of using ASP.NET Web API over WCF services is that the client does need to consume a proxy in order to make a call to Web API service. This facilitates decoupling between the client and the Web API service.

The following diagram explains the architecture of calling Web API using a Client Application.


HttpClient class is used for Making Calls to WEB API and perform Http Operations.
The Managed Client Application. Desktop, ASP.NET, ASP.NET MVC Using C#
WEB API Controller
@DotNetCurry.com
Data Access Repository
ADO .NET EF
Database
WEB API Service

The above diagram explains the basic arrangement of a Web API Service and the client application.

- The API Service communicates with Database for writing and reading data and uses ADO.NET EntityFramework to communicate with the database.
- The Data Access Repository, decouples the ADO.NET EntityFramework from the ApiController class, using the simple implementation of the repository pattern, implemented using Unity.WebApi framework.
- The WEB API Controller class calls the Data Access Repository for performing Read/Write operations.
- The Managed Client Application, uses HttpClient class for performing an asynchronous calls to the Web API.

HttpClient

This is the base class used for making an asynchronous calls to the remote service repository using its URL. This class is present under the System.Net.Http namespace and provide methods like GetAsync (), PostAsync (), PutAsync () and DeleteAsync ().

The Implementation

Step 1: Open Visual Studio and create a Blank solution, name this solution as 'MVC_ClientWebAPI'.

In this solution, add a new ASP.NET Web API Project of the name APIService and click OK.

This will create a Web API project with the necessary folders for Models, Controllers, App_Start, App_Data, etc.

Step 2: In the App_Data folder, add a new Sql Server database of name 'Application.mdf'.

In this database add the EmployeeInfo table using the following script.

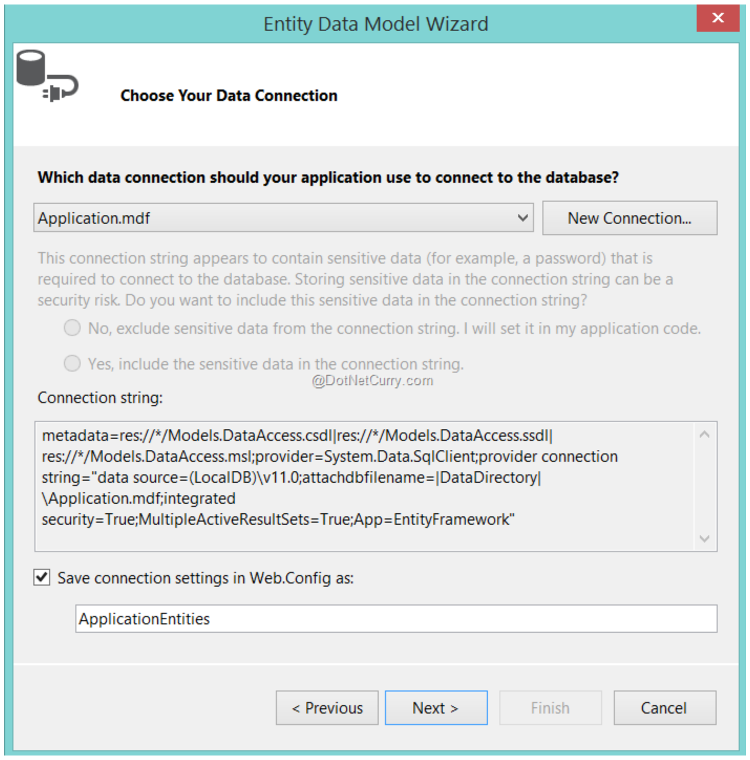
```
CREATE TABLE [dbo].[EmployeeInfo] (

    [EmpNo]       INT      IDENTITY (1, 1) NOT NULL,

    [EmpName]     VARCHAR (50) NOT NULL,

    [DeptName]    VARCHAR (50) NOT NULL,

    [Designation] VARCHAR (50) NOT NULL,

    [Salary]      DECIMAL (18) NOT NULL,

    PRIMARY KEY CLUSTERED ([EmpNo] ASC)

);
```

In this table add the sample data as shown in the following script

```
INSERT INTO [dbo].[EmployeeInfo] ([EmpNo], [EmpName], [DeptName], [Designation],
[Salary]) VALUES (1, N'MS', N'MS IT Services', N'Director', CAST(780000 AS Decimal(18,
0)))

INSERT INTO [dbo].[EmployeeInfo] ([EmpNo], [EmpName], [DeptName], [Designation],
[Salary]) VALUES (2, N'LS', N'Administration', N'Manager', CAST(1556000 AS Decimal(18,
0)))

INSERT INTO [dbo].[EmployeeInfo] ([EmpNo], [EmpName], [DeptName], [Designation],
[Salary]) VALUES (3, N'TS', N'Administration', N'Dy. Manager', CAST(45000 AS
Decimal(18, 0)))
```

Step 3: In the Models folder add a new ADO.NET EF. Name this EF as DataAccess. This will start a Wizard. In this wizard, select the EF Designer from Database,

(Note: Since the article uses VS2013 with Update 4, the EF wizard shows EF Designer from Database, else the option may be Generate from Database). The wizard selects Application.mdf, as shown in the following image:



Which data connection should your application use to connect to the database?

Application.mdf

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

- No, exclude sensitive data from the connection string. I will set it in my application code.
- Yes, include the sensitive data in the connection string.

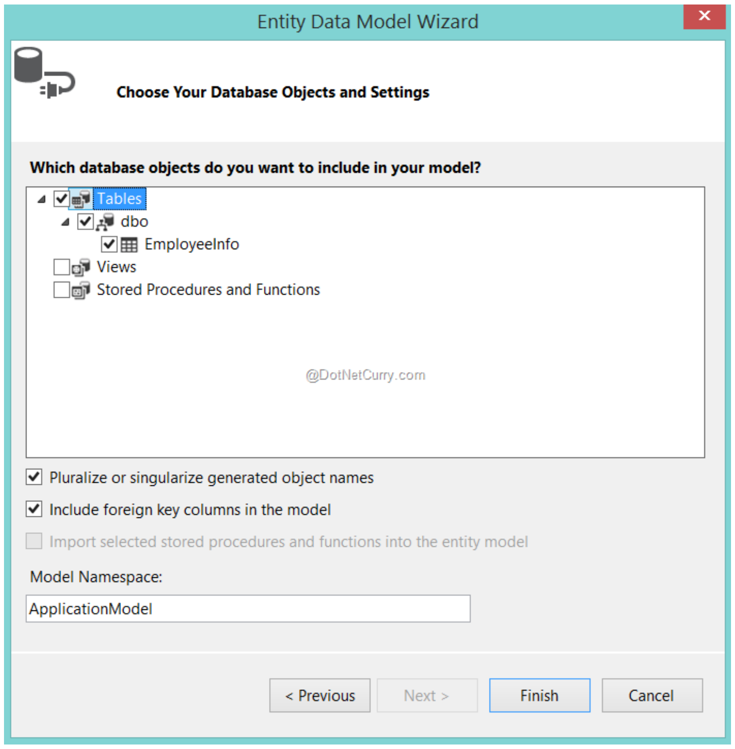
Connection string:

```
metadata=res://*/Models.DataAccess.csdl|res://*/Models.DataAccess.ssdl|
res://*/Models.DataAccess.msl;provider=System.Data.SqlClient;provider connection
string="data source=(LocalDB)\v11.0;attachdbfilename=|DataDirectory|
\Application.mdf;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
```

☑ Save connection settings in Web.Config as:

ApplicationEntities

In the next step of the wizard, select EmployeeInfo table as shown in the following image:


Entity Data Model Wizard

Choose Your Database Objects and Settings

Which database objects do you want to include in your model?

- [✓] Tables
  - [✓] dbo
    - [✓] EmployeeInfo
- [ ] Views
- [ ] Stored Procedures and Functions

- [✓] Pluralize or singularize generated object names
- [✓] Include foreign key columns in the model
- [ ] Import selected stored procedures and functions into the entity model

Model Namespace: ApplicationModel

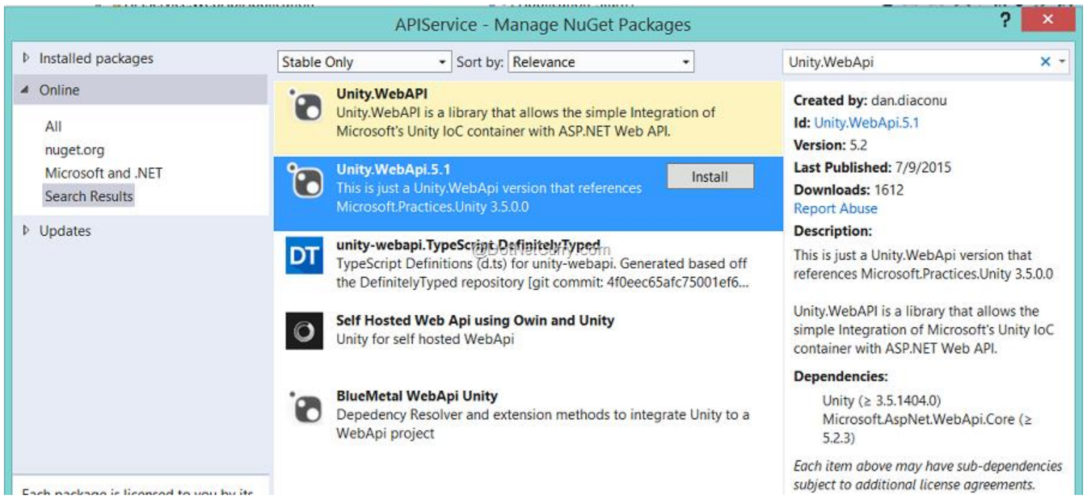
After completion of this wizard, the table mapping will be generated. This creates EmployeeInfo class and ApplicationEntities class in the Models folder.

-----→Build the project.

Step 4: Since we will be using a simple Repository pattern to decouple the Web API Controller class from the ADO.NET EF, we need Unity framework for dependency injection for the Web API using constructor injection.

To implement this, in the project add Unity.WebApi.5.1 (or whatever version is latest) NuGet package using the Package Manager window.

To implement this, right-click on the project and select Manage NuGet Packages. From the NuGet Packages window, select Unity.WebApi.5.1 as shown in the following image:


APIService - Manage NuGet Packages

Unity.WebAPI
Unity.WebAPI is a library that allows the simple Integration of Microsoft's Unity IoC container with ASP.NET Web API.

Unity.WebApi.5.1
This is just a Unity.WebApi version that references Microsoft.Practices.Unity 3.5.0.0

unity-webapi.TypeScript.DefinitelyTyped
TypeScript Definitions (d.ts) for unity-webapi. Generated based off the DefinitelyTyped repository [git commit: 4f0eec65afc75001ef6...

Self Hosted Web Api using Owin and Unity
Unity for self hosted WebApi

BlueMetal WebApi Unity
Depedency Resolver and extension methods to integrate Unity to a WebApi project

Created by: dan.diaconu
Id: Unity.WebApi.5.1
Version: 5.2
Last Published: 7/9/2015
Downloads: 1612
Report Abuse
Description:
This is just a Unity.WebApi version that references Microsoft.Practices.Unity 3.5.0.0

Unity.WebAPI is a library that allows the simple Integration of Microsoft's Unity IoC container with ASP.NET Web API.

Dependencies:
Unity (≥ 3.5.1404.0)
Microsoft.AspNet.WebApi.Core (≥ 5.2.3)

Each item above may have sub-dependencies subject to additional license agreements.

Each package is licensed to you by its

Click on the install button. This will add Unity.WebApi reference in the project references. The App_Start folder will be added with the UnityConfig.cs file with UnityConfig class in it. We will modify this class in the coming steps.

Step 5: We need to create a separate class to implement the data access logic.

- In the project, add a new folder of name 'DataAccessRepository'.
- In this folder, add a new interface file with the code as shown as following:

```
public interface IDataAccessRepository<TEntity, in TPrimaryKey> where TEntity : class

{

    IEnumerable<TEntity> Get();

    TEntity Get(TPrimaryKey id);

    void Post(TEntity entity);

    void Put(TPrimaryKey id,TEntity entity);

    void Delete(TPrimaryKey id);

}
```

This generic interface contains methods for Get, Post, Put and Delete operations. This interface is typed to the following parameters:

- TEntity: This is the class type, represents the entity class used during Read/Write operations.
- TPrimaryKey: This is an input parameter used during Reading of specific record and deleting it.

Step 5: In the DataAccessRepository folder, add a new class of clsDataAccessRepository with the following code:

```
using System.Collections.Generic;

using System.Linq;

using APIService.Models;


using Microsoft.Practices.Unity;


namespace APIService.DataAccessRepository

{

    public class clsDataAccessRepository : IDataAccessRepository<EmployeeInfo, int>

    {

        //The dendency for the DbContext specified the current class.

        [Dependency]
        public ApplicationEntities ctx { get; set; }

        //Get all Employees
        public IEnumerable<EmployeeInfo> Get()
        {
            return ctx.EmployeeInfoes.ToList();
        }
        //Get Specific Employee based on Id
        public EmployeeInfo Get(int id)
        {
            return ctx.EmployeeInfoes.Find(id);
        }

        //Create a new Employee
        public void Post(EmployeeInfo entity)
        {
            ctx.EmployeeInfoes.Add(entity);
            ctx.SaveChanges();
        }
        //Update Exisitng Employee
        public void Put(int id, EmployeeInfo entity)
        {
            var Emp = ctx.EmployeeInfoes.Find(id);
            if (Emp != null)
            {
                Emp.EmpName = entity.EmpName;
                Emp.Salary = entity.Salary;
                Emp.DeptName = entity.DeptName;
                Emp.Designation = entity.Designation;
                ctx.SaveChanges();
            }
        }
        //Delete an Employee based on Id
        public void Delete(int id)
        {
            var Emp = ctx.EmployeeInfoes.Find(id);
            if (Emp != null)
            {
                ctx.EmployeeInfoes.Remove(Emp);
                ctx.SaveChanges();
            }
        }
    }
}
```

The above class code is specified with the ApplicationEntities DbContext injection using [Dependency] attribute on it. This eliminates need of creating instance using *new* keyword. The class contains methods for performing Read/Write operations using ADO.NET EF methods.

Step 6: To add the above code class in the Unity Container, we need to change the UnityConfig class from the App_Start folder as shown in the following code:

```
public static class UnityConfig
{
    public static void RegisterComponents()
    {
        var container = new UnityContainer();

        container.RegisterType<IDataAccessRepository<EmployeeInfo,int>,
clsDataAccessRepository>();

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

The RegisterType () method accepts the interface and class for registering in the Unity container. The UnityDependencyResolver wraps the container object of the type UnityContainer in it. This resolves the dependencies registered inside the container. In our case, we have registered IDataAccessRepository interface and clsDataAccessRepository class.

Open Global.asax, and add the following line in it:

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    UnityConfig.RegisterComponents();
    GlobalConfiguration.Configure(WebApiConfig.Register);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}
```

The above line of code calls the RegisterComponents () method in the Application_Start event so that all dependency must be registered in the beginning of the application.

Step 7: In the Controllers, add a new Empty Web API Controller of the name EmployeeInfoAPIController. In this controller, add the following code:

```
using System.Collections.Generic;
using System.Net;
using System.Web.Http;
using APIService.Models;
using APIService.DataAccessRepository;
using System.Web.Http.Description;

namespace APIService.Controllers
{
    public class EmployeeInfoAPIController : ApiController
    {
        private IDataAccessRepository<EmployeeInfo, int> _repository;
        //Inject the DataAccessRepository using Construction Injection
        public EmployeeInfoAPIController(IDataAccessRepository<EmployeeInfo, int> r)
        {
            _repository = r;
        }
        public IEnumerable<EmployeeInfo> Get()
        {
            return _repository.Get();
        }

        [ResponseType(typeof(EmployeeInfo))]
        public IHttpActionResult Get(int id)
        {
            return Ok (_repository.Get(id));
        }

        [ResponseType(typeof(EmployeeInfo))]
        public IHttpActionResult Post(EmployeeInfo emp)
        {
            _repository.Post(emp);
            return Ok(emp);
        }

        [ResponseType(typeof(void))]
        public IHttpActionResult Put(int id ,EmployeeInfo emp)
        {
            _repository.Put(id,emp);
            return StatusCode(HttpStatusCode.NoContent);
        }
        [ResponseType(typeof(void))]
        public IHttpActionResult Delete(int id)
        {
            _repository.Delete(id);
            return StatusCode(HttpStatusCode.NoContent);
        }
    }
}
```

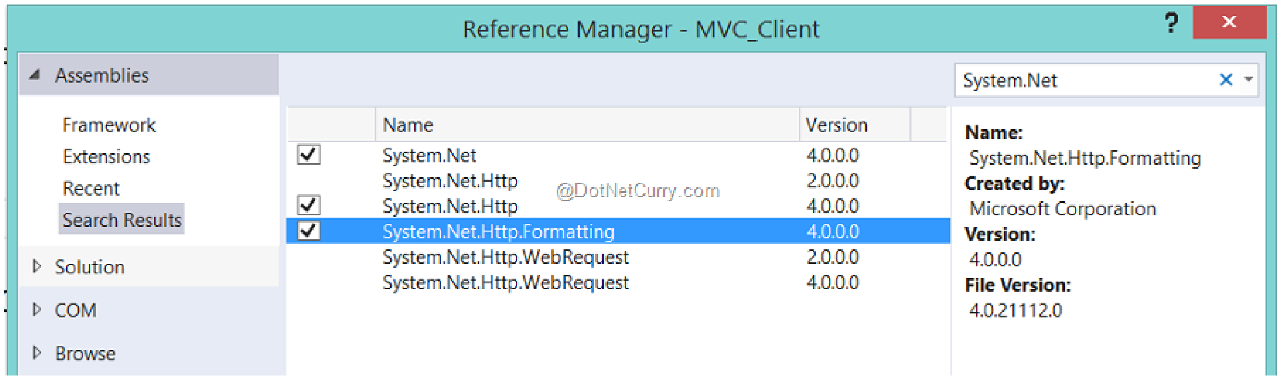
In the above controller, the clsDataAccessRepository object is injected using Constructor Injection. The controller contains methods for performing Http GET, POST, PUT and DELETE operations.

Implementing the ASP.NET MVC Client Applciation

Step 1: In the same solution, add a new ASP.NET MVC application of name 'MVC_Client'.

You can also select Empty MVC application. If you select an empty mvc app, then in that case reference for Json.NET must be added explicitly using NuGet Package Manager. This is for JSON Serialization and Deserialization of the Entity object during Read/Write operation.

To make Http calls using C#, we need to add references for System.Net Http.



	Name	Version
✓	System.Net	4.0.0.0
	System.Net.Http	2.0.0.0
✓	System.Net.Http	4.0.0.0
✓	System.Net.Http.Formatting	4.0.0.0
	System.Net.Http.WebRequest	2.0.0.0
	System.Net.Http.WebRequest	4.0.0.0

Name: System.Net.Http.Formatting
Created by: Microsoft Corporation
Version: 4.0.0.0
File Version: 4.0.21112.0

Note: The NuGet Package for Json.Net and references for System.Net.Http is not required if an ASP.NET MVC ready template is chosen.

Step 2: In this project in the Models folder add the following EmployeeInfo class.

```
public class EmployeeInfo
{
    public int EmpNo { get; set; }
    public string EmpName { get; set; }
    public decimal Salary { get; set; }
    public string DeptName { get; set; }
    public string Designation { get; set; }
}
```

This is the same class available on the API Service side. We need this for the Serialization and Deserialization during Read/Write operations.

Step 3: In the Controllers folder, add a new Empty MVC Controller of the name EmployeeInfoController. In this controller, add the following code:

```
using Newtonsoft.Json;


namespace MVC_Client.Controllers
{
    public class EmployeeInfoController : Controller
    {

        HttpClient client;
        //The URL of the WEB API Service
        string url = "http://localhost:60143/api/EmployeeInfoAPI";

        //The HttpClient Class, this will be used for performing
        //HTTP Operations, GET, POST, PUT, DELETE
        //Set the base address and the Header Formatter
        public EmployeeInfoController()
        {
            client = new HttpClient();
            client.BaseAddress = new Uri(url);
            client.DefaultRequestHeaders.Accept.Clear();
            client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
        }

        // GET: EmployeeInfo
        public async Task<ActionResult> Index()
        {
            HttpResponseMessage responseMessage = await client.GetAsync(url);
            if (responseMessage.IsSuccessStatusCode)
            {
               var responseData =   responseMessage.Content.ReadAsStringAsync().Result
;

               var Employees =
JsonConvert.DeserializeObject<List<EmployeeInfo>>(responseData);

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

        public ActionResult Create()
        {
            return View(new EmployeeInfo());
        }

        //The Post method
        [HttpPost]
        public async Task<ActionResult> Create(EmployeeInfo Emp)
        {

           HttpResponseMessage responseMessage =  await
client.PostAsJsonAsync(url,Emp);
           if (responseMessage.IsSuccessStatusCode)
           {
               return  RedirectToAction("Index");
           }
```

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

            public async Task<ActionResult> Edit(int id)
            {
                HttpResponseMessage responseMessage = await client.GetAsync(url+"/"+id);
                if (responseMessage.IsSuccessStatusCode)
                {
                    var responseData = responseMessage.Content.ReadAsStringAsync().Result;

                    var Employee = 
JsonConvert.DeserializeObject<EmployeeInfo>(responseData);

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

            //The PUT Method
            [HttpPost]
            public async Task<ActionResult> Edit(int id,EmployeeInfo Emp)
            {

                HttpResponseMessage responseMessage = await client.PutAsJsonAsync(url+"/"
+id, Emp);
                if (responseMessage.IsSuccessStatusCode)
                {
                    return RedirectToAction("Index");
                }
                return RedirectToAction("Error");
            }
```

```
        public async Task<ActionResult> Delete(int id)
        {
            HttpResponseMessage responseMessage = await client.GetAsync(url + "/" +
id);
            if (responseMessage.IsSuccessStatusCode)
            {
                var responseData = responseMessage.Content.ReadAsStringAsync().Result;

                var Employee =
JsonConvert.DeserializeObject<EmployeeInfo>(responseData);

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

        //The DELETE method
        [HttpPost]
        public async Task<ActionResult> Delete(int id, EmployeeInfo Emp)
        {

            HttpResponseMessage responseMessage =await client.DeleteAsync(url + "/" +
id);
            if (responseMessage.IsSuccessStatusCode)
            {
                return RedirectToAction("Index");
            }
            return RedirectToAction("Error");
        }
    }
```

```
}
```

The above controller class has the following features:

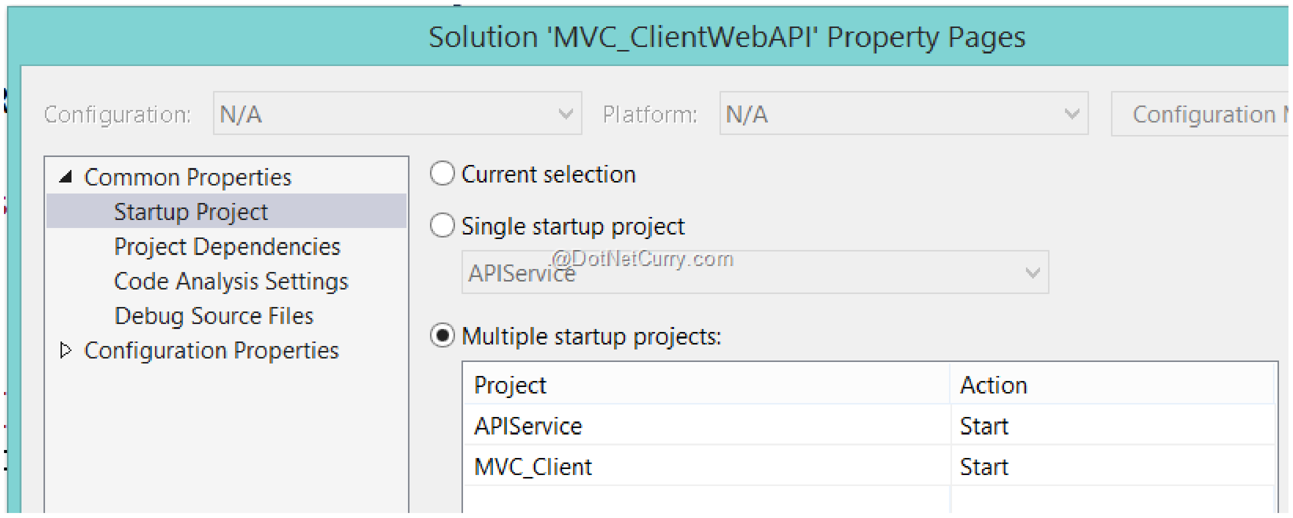
- The constructor of the class instantiates the HttpClient class. This instance is set with the properties like BaseAddress, this represents the remote Web API URL with which the Read/Write operations are performed.
- The DefaultRequestHeaders is set to accept the media type formatter for json data.
- The Index () method uses the GetAsyc() method to receive the HttpResponseMessage. Since the response message contains the Json string contents, this content can be read using ReadAsStringAsync() function.
- The string contents are DeSerialized to the List of Employees using DeserializeObject () method of JsonConvert class. Since this call is asynchronous, the Index() method returns Task<T> object where T is ActionResult.
- Create() function with HttpPost uses PostJsonAsAsync() method to make the asynchronous post of the EmployeeInfo object to the Web API Service.
- Similarly Edit() method uses PutAsJsonAsync() method and Delete method uses DeleteAsync() methods for performing respective operations using Web API.

Step 4: To generate UI for the application, scaffold views from the Index, Create, Edit, Delete (all HttpGet) methods. This adds Index.cshtml, Create.cshtml, Edit.cshtml and Delete.cshtml in the EmployeeInfo folder of the Views folder.

Step 5: To run the Index.cshtml, change the RouteConfig class from RouteConfig.cs file of the App_Start folder.

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "EmployeeInfo", action = "Index", id =
UrlParameter.Optional }
        );
    }
}
```

Step 6: To run the API Service and MVC Client project, right-click on the solution and set the multiple startup project as shown in the following image:


Solution 'MVC_ClientWebAPI' Property Pages

Configuration: N/A Platform: N/A Configuration M

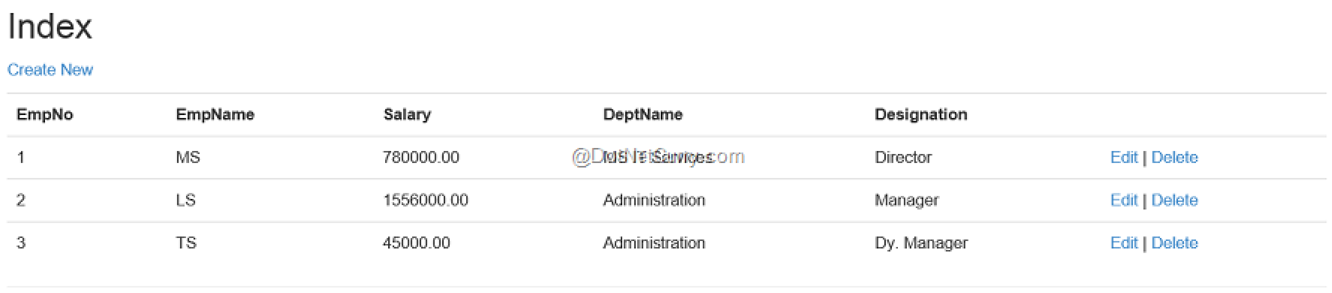
- Common Properties
  - Startup Project
  - Project Dependencies
  - Code Analysis Settings
  - Debug Source Files
- Configuration Properties

Current selection [✓ Multiple startup projects:]

Single startup project: APIService

Project	Action
APIService	Start
MVC_Client	Start

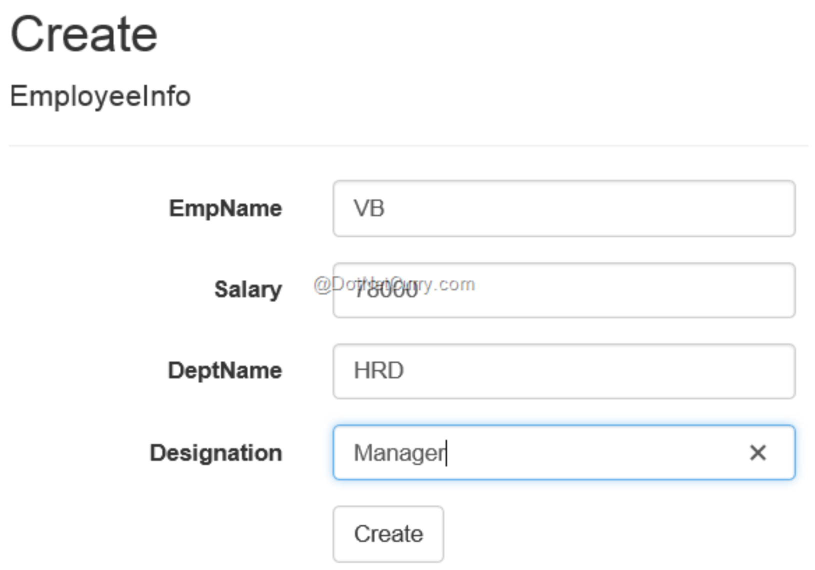
Run the application, the Index view of the Employee will be displayed as shown below:


Index

Create New

EmpNo	EmpName	Salary	DeptName	Designation	
1	MS	780000.00	MS IT Services	Director	Edit | Delete
2	LS	1556000.00	Administration	Manager	Edit | Delete
3	TS	45000.00	Administration	Dy. Manager	Edit | Delete

Click on the Create New link, and the Create View will be displayed. Enter the Employee data in it as shown in the following image:

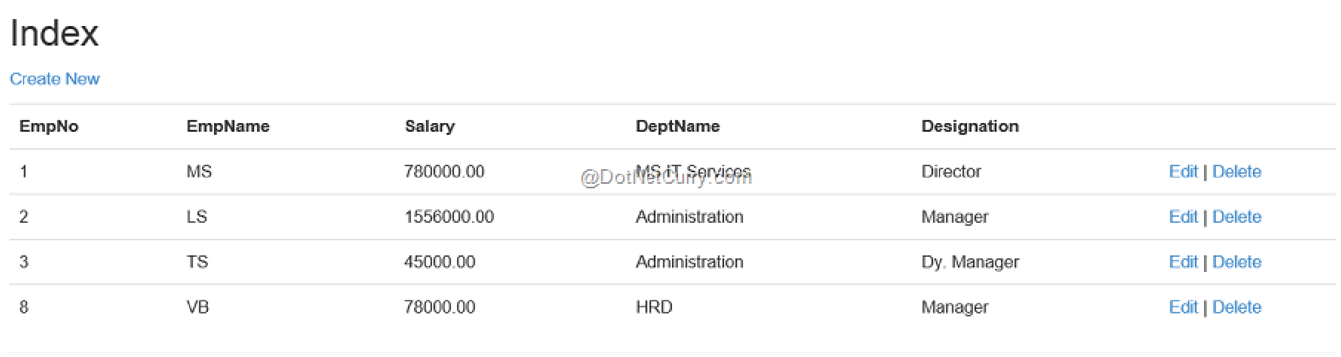

Create

EmployeeInfo

EmpName	VB
Salary	78000
DeptName	HRD
Designation	Manager

Create

Click on the 'Create' button, the control will be redirected to the Index view with the record added as shown in the following image.



Index

Create New

EmpNo	EmpName	Salary	DeptName	Designation	
1	MS	780000.00	MS IT Services	Director	Edit | Delete
2	LS	1556000.00	Administration	Manager	Edit | Delete
3	TS	45000.00	Administration	Dy. Manager	Edit | Delete
8	VB	78000.00	HRD	Manager	Edit | Delete

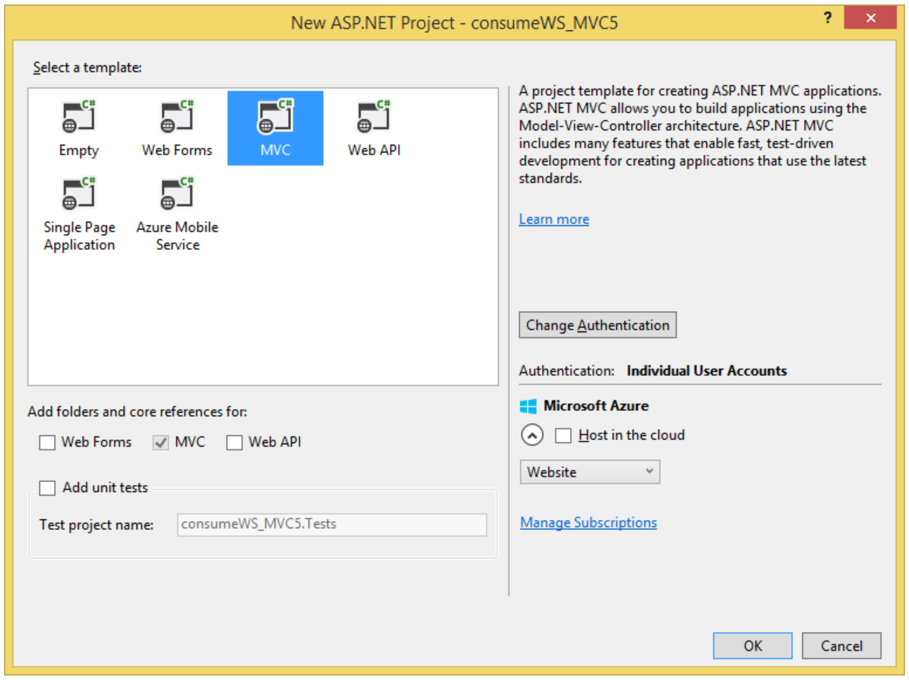
Conclusion: We saw how the HTTPClient class provides useful methods to call an ASP.NET Web API service asynchronously from the managed clients like WPF or ASP.NET MVC.

End Painful Example…

Next Example (somewhat incomplete)

A RESTful Web Service, e.g. Web API service, and you want to create a new MVC application that consume this Web Service. I designed a simple way to achieve the goal.

From Visual Studio wizard create a new MVC 5 Project.



New ASP.NET Project - consumeWS_MVC5

Select a template:

Empty, Web Forms, [✓ MVC], Web API, Single Page Application, Azure Mobile Service

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Learn more

Change Authentication

Authentication: **Individual User Accounts**

Microsoft Azure

☐ Host in the cloud

Website

Manage Subscriptions

Add folders and core references for:

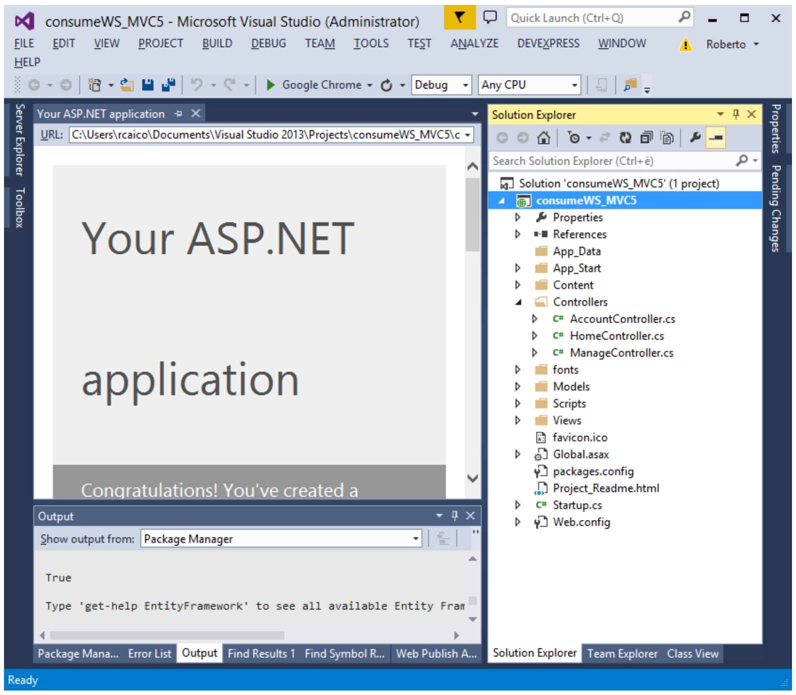
☐ Web Forms ☑ MVC ☐ Web API

☐ Add unit tests

Test project name: consumeWS_MVC5.Tests

OK Cancel

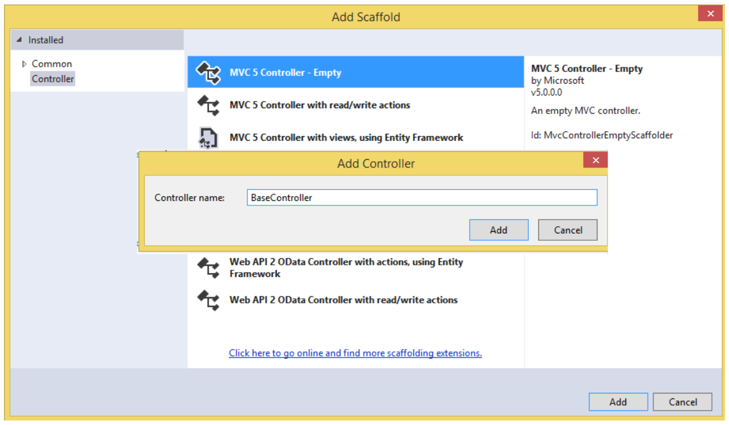
By default you'll find 3 standard controllers: Account, Manage and Home.



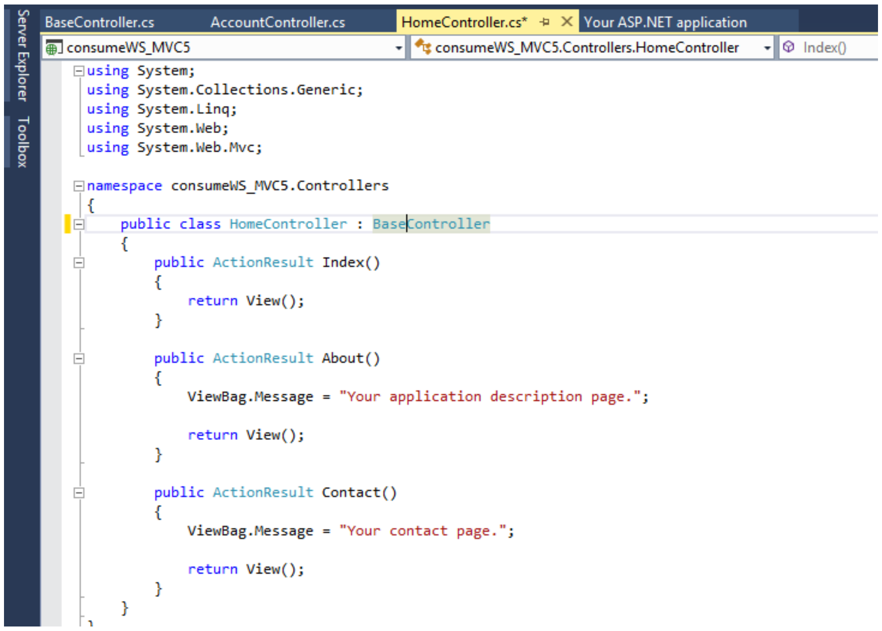
```
True
Type 'get-help EntityFramework' to see all available Entity Fram
```

And all inherits from Controller.

Now create a new MVC 5 controller and name it: "BaseController".

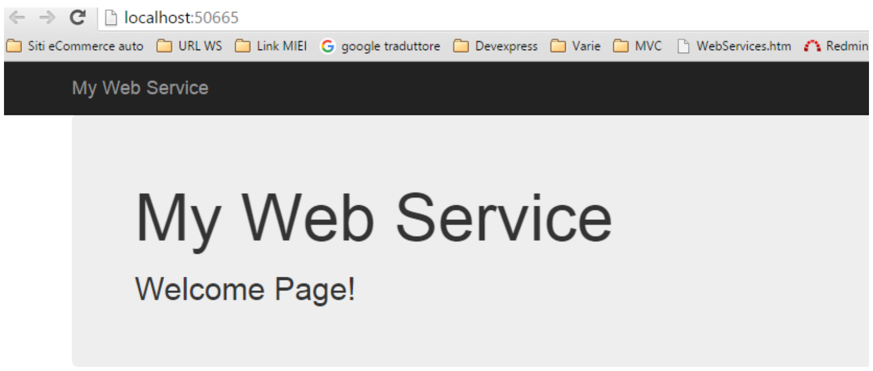

Add Scaffold
Installed
Common
Controller
MVC 5 Controller - Empty
MVC 5 Controller with read/write actions
MVC 5 Controller with views, using Entity Framework
Web API 2 OData Controller with actions, using Entity Framework
Web API 2 OData Controller with read/write actions
Click here to go online and find more scaffolding extensions.
MVC 5 Controller - Empty
by Microsoft
v5.0.0.0
An empty MVC controller.
Id: MvcControllerEmptyScaffolder
Add Controller
Controller name: BaseController
Add
Cancel

Change the generalization of the other controllers and implement the BaseController. I have shown it in the following screenshot under Home Controller,



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace consumeWS_MVC5.Controllers
{
    public class HomeController : BaseController
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult About()
        {
            ViewBag.Message = "Your application description page.";

            return View();
        }

        public ActionResult Contact()
        {
            ViewBag.Message = "Your contact page.";

            return View();
        }
    }
```

Suppose now there exist a web API service,



My Web Service

Welcome Page!

And you want contact this for get your Business Object Info as :

- **Products Detail:** localhost:50665\api\Products
- **Orders Detail:** localhost:50665\api\Orders
- **Customers Detail:** localhost:50665\api\Customers
- **Invoice Detail:** localhost:50665\api\Document\GetInvoice?num=1

So, I think I found a very easy way,

I created Base Controller, a generic method (**GetWSObject**) for contacting the WebService from all Controller and returned my desired object as a dynamic object:

```
public async Task < T > GetWSObject < T > (string uriActionString)
{
    T returnValue =
        default (T);
    try
    {
        using(var client = new HttpClient())
        {
            client.BaseAddress = new Uri(@ "http://localhost:50665/");
              client.DefaultRequestHeaders.Accept.Clear();
              client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
              HttpResponseMessage response = await client.GetAsync(uriActionString);
              response.EnsureSuccessStatusCode();
              returnValue = JsonConvert.DeserializeObject < T > (((HttpResponseMessage) response).Content.ReadAsStringAsync().Result);
          }
          return returnValue;
      }
      catch (Exception e)
      {
          throw (e);
      }
  }
```

The method takes the uriActionString as an input parameter and returns the result dynamically deserialized as my business model object .

So, for example, if you want to take the detail for Product with id=11 and show this in your MVC application in the HomeControllers (that implement the BaseController) you can create the following ActionResult:

```
public async Task < ActionResult > ViewProduct(int Id)
{
    ViewBag.Message = "Your products page.";
    Product product = new Product();
    string urlAction = String.Format("/api/Products/{0}", Id);
    product = await GetWSObject < Product > (urlAction);
    return View(product);
}
```

With 2 lines you get the result and you have a Product object to pass (e.g.) in your View as a viewmodel.

Obviously, with the same modalities you can contact any other method of the Web Service, from any other Controller; simply change the urlAction string and call the GetWSObject method passing the desired result object type (<Product>).

References

1. Consuming a Web API Asynchronously in ASP.NET MVC or WPF: http://www.dotnetcurry.com/aspnet/1192/aspnet-web-api-async-calls-mvc-wpf
    a. Both the Publishing API and MVC client
    b. Completed Project: webapi-async-wpf-mvc: https://github.com/dotnetcurry/webapi-async-wpf-mvc
2. Consuming Web Services In ASP.NET MVC 5 Application: http://www.c-sharpcorner.com/UploadFile/653eb2/consuming-web-services-in-Asp-Net-mvc-5-application/
3. ANCIENT – Using XML : No client side - RESTful XHTML - RESTful Services With ASP.NET MVC: https://msdn.microsoft.com/en-us/magazine/dd943053.aspx