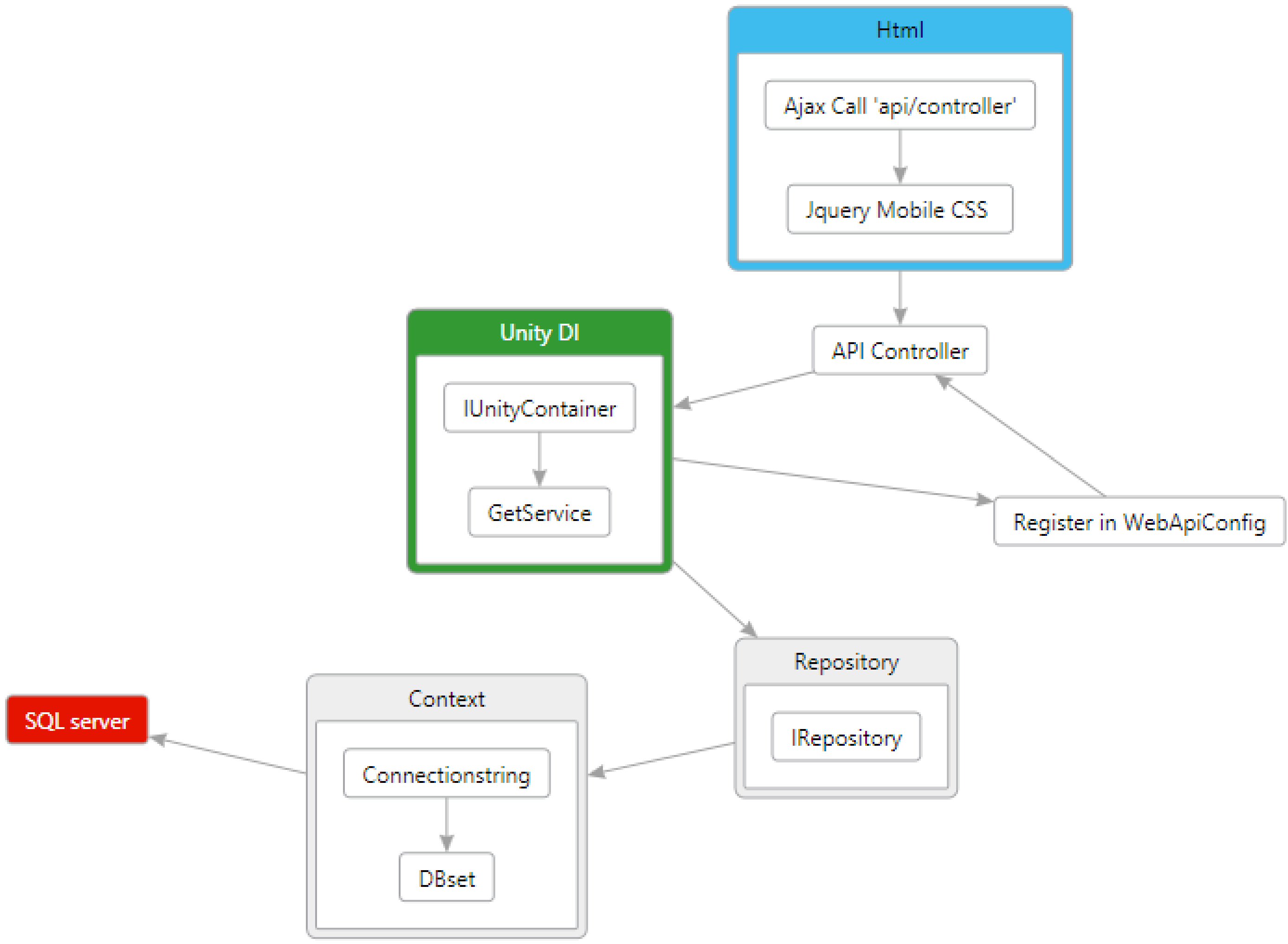
Create Asp.Net Web API with Unity DI container in VS2013

Introduction

This project will test how to use Unity in Visual Studio 2013 web api development see image below.



Main Tasks

1, Create an Asp.net web api in VS 2013

Create a new project and select empty asp.net web application then click next, select empty and tick web api then click next. A pure asp.net web api template is created.

2, create a virtual directory in IIS web server

Now we need to host this web api in IIS web server. Select web option in the property of the api project, select local IIS and then click create a virtual directory button. Press F5 to run the project and check if web page is displayed properly. If yes, It proves that this web api now is hosted in IIS web server. Now select app pool and defaultwebsite session in IIS web server, right click and select advanced setting to find out the identity field and then select local system account. It means we now use NT Authority/System account to access to this web server. This account should also be the SQL server login user to allow web server connect to SQL server via network. Therefore, using the following t-SQL code to create such a new user in database and then grant db_owner role to this account. After we press F5 in visual studio 2013, NT Authority/System account is used to accsess to web server and SQL server.

```
EXEC sp_addrolemember 'db_owner', 'NT AUTHORITY\SYSTEM'

CREATE USER [NT AUTHORITY\SYSTEM]

        WITHOUT LOGIN

        WITH DEFAULT_SCHEMA = dbo

GO



GRANT CONNECT TO  [NT AUTHORITY\SYSTEM]

GRANT CREATE TABLE

TO  [NT AUTHORITY\SYSTEM]

GO
```

3, Add model classes

Add a product class with id, name, and price fields in model folder.

```
public class Product

            {

                public int Id { get; set; }

                public string Name { get; set; }

                public decimal Price { get; set; }

            }
```

Download Entityframework ver 6 and then add a context class to create a new code first database and connect the project to this product database

```
public class ProductsContext : DbContext

            {

                public ProductsContext() : base("name=Pcontext")

                {

                }

                public DbSet< product> Products { get; set; }

            }
```

Code first action also creates a migration history table in the database and also create a products table that name is the same as the name from ProductsContext. Press F5 to run and call 'api/products' web service resource, this new database and tables will be created first via EF framework code first. If we want to change something in this database, we use update database and enable migration options in PM commands to do the transactions.

Now add a SQL database connectionstring called 'Pcontext' in the web.config file of this web api project as below

```
            < connectionstrings>

            < add name="Pcontext" connectionstring="Data
Source=LIZXINYOU\SQLEXPRESS;Initial Catalog=Products;Integrated Security=True;"
providername="System.Data.SqlClient" />

            < /connectionstrings>
```

4, Create classes based on repository design pattern

We know we will not allow mvc or api controller classes in MVC or Web API to access to the database diectly. We need to create a DAL. Therefore, an abstraction data access layer is created as below

```
public interface IProductRepository

          {

            IEnumerable< product> GetAll();

            Product GetById(int id);

            void Add(Product product);

          }
```

create a concrete repository class to implement this abstraction data access layer

```
public class ProductRepository : IDisposable, IProductRepository
```

```
{
    private ProductsContext db = new ProductsContext();
    public IEnumerable< product> GetAll()
    {
        return db.Products;
    }
    public Product GetById(int id)
    {
        return db.Products.FirstOrDefault(p => p.Id == id);
    }
    public void Add(Product product)
    {
        db.Products.Add(product);
        db.SaveChanges();
    }
    protected void Dispose(bool disposing)
    {
        if (disposing)
        {
                if (db != null)

                {

                    db.Dispose();

                    db = null;

                }

            }

        }

        public void Dispose()

        {

            Dispose(true);

            GC.SuppressFinalize(this);

        }

    }
```

5, Create a proxy data access layer -- API controller

Now we will expose data access layer to clients via creating an API controller productscontroller to return data resources fromm database by this asp.net Web api.

```


    public class ProductsController : ApiController

        {
```

```
        private IProductRepository _repository;

        public ProductsController(IProductRepository repository)

        {

            _repository = repository;

        }

        public IEnumerable< product> Get()

        {

            return _repository.GetAll();

        }

        public IHttpActionResult Get(int id)

        {

            var product = _repository.GetById(id);

            if (product == null)

            {

                return NotFound();

            }

            return Ok(product);

        }

    }
```

But there is a problem when we do this, we can not complie the api appication because the application can not create the controller directly without having a default parameterless constructor. Web API creates the controller when it routes the request, and Web API doesn't know anything about the injected abstract IProductRepository interface. Therefore, it can not find a default constructor afer we inject the repository interface in. We need to introduce the web api dependency resolver to fix this problem.

6, Web API dependency resolver

create two abstract interfaces as below

```
public interface IDependencyResolver : IDependencyScope, IDisposable

{

    IDependencyScope BeginScope();

}

public interface IDependencyScope : IDisposable

{

    object GetService(Type serviceType);

    IEnumerable< object> GetServices(Type serviceType);

}
```

When Web API creates a controller instance, it first calls IDependencyResolver.GetService, passing in the controller type(we can use this extensibility hook to create the controller, resolving any dependencies). If GetService returns null, Web API looks for a parameterless constructor on the controller class.

this interface is really designed to act as bridge between Web API and existing IoC containers. Unity IoC container is a software component that is responsible for managing dependencies. We register types with the container, and then use the container to create objects. The container automatically figures out the dependency relations. Many IoC containers also allow us to control things like object lifetime and scope. Now we implement this abstract interface bridge called UnityResolver as below.

```
public class UnityResolver : IDependencyResolver

{

    protected IUnityContainer container;

    public UnityResolver(IUnityContainer container)

    {

        if (container == null)

        {

            throw new ArgumentNullException("container");

        }

        this.container = container;

    }

    public object GetService(Type serviceType)

    {

        try

        {
```

```
        return container.Resolve(serviceType);
    }
    catch (ResolutionFailedException)
    {
        return null;
    }
}
public IEnumerable< object> GetServices(Type serviceType)
{
    try
    {
        return container.ResolveAll(serviceType);
    }
    catch (ResolutionFailedException)
    {
        return new List< object>();
    }
}
public IDependencyScope BeginScope()
```

```
        {

            var child = container.CreateChildContainer();

            return new UnityResolver(child);

        }

        public void Dispose()

        {

            container.Dispose();

        }

    }
```

UnityContainer is injected into this class. Class allows this container to resolve the data types and creates objects for those data types which can be used as a data source in controller. So top level controller has such lower level data types to consume.

Therefore, GetServices method returns a series of source objects for controller.

7, Configuring the Dependency Resolver

We have to hook this bridge for controller to bring the source objects in as below

```
      public static void Register(HttpConfiguration config)

        {

             IUnityContainer container = new UnityContainer();

            container.RegisterType< iproductrepository, productrepository>(new
 HierarchicalLifetimeManager());

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

when the application gets started, container creates source objects for controller constructor. Now if the controller is called, it recognizes the source objects from container. Controller object without parameterless constructor is created succesfully.

8, Create Jquery Mobile Index.html page to consume this API

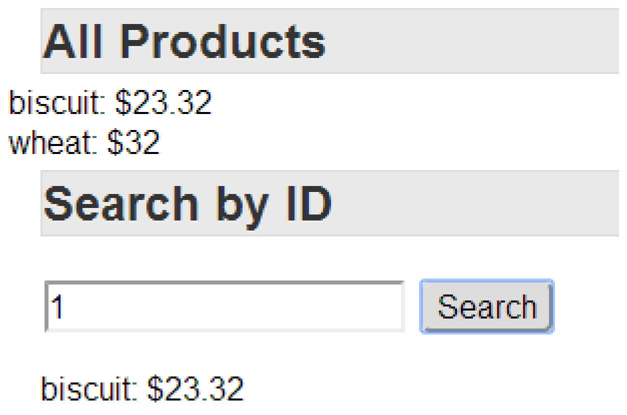
This Asp.Net Web API now can be consumed by a pure html page with the assistance of Jquery getJSON call see the code below.

```
   < div data-role="page" data-theme="a">

   < div role="main" class="ui-content">

   < div>

   < h2 class="ui-bar-a">All Products

   < ul id="products" data-role="listview" class="ui-listview" />

 < /div>

   < div>

   < h2 class="ui-bar-a">Search by ID

   < input type="text" name="text-1" id="prodId">

   < button onclick="find();" class="ui-btn-a ui-btn-corner-all  ui-icon-arrow-r">Search

   < p id="product" />

< /div>

    < script>

      var uri = 'api/products';
```

```
$(document).ready(function () {

    $.getJSON(uri)

        .done(function (data) {

            $.each(data, function (key, item) {

                $('< li>', { text: formatItem(item) }).appendTo($('#products'));

            });

        });

});



function formatItem(item) {

    return item.Name + ': $' + item.Price;

}



function find() {

    var id = $('#prodId').val();

    $.getJSON(uri + '/' + id)

        .done(function (data) {

            $('#product').text(formatItem(data));

        })
```

```
                    .fail(function (jqXHR, textStatus, err) {

                        $('#product').text('Error: ' + err);

                    });

                }

            < /script>

        < /div>

    < /div>
```

This resut is as the following image shown


All Products

biscuit: $23.32
wheat: $32

Search by ID

1 Search

biscuit: $23.32

Summary

Visual Studio 2013 enables Dependency injection feature that will allow us to develop Unity based Web API easily and allow us to use pure index.html page to consume this nice Web API resource with the helps of JavaScript Ajax call.