Developing Business Applications with Silverlight 3 ,Prism and Ria Services

This series of articles is designed to give the reader an overview of using the combination of Silverlight 3 and Prism to deliver rich web business applications. It will additionally discuss using Ria Data Services.

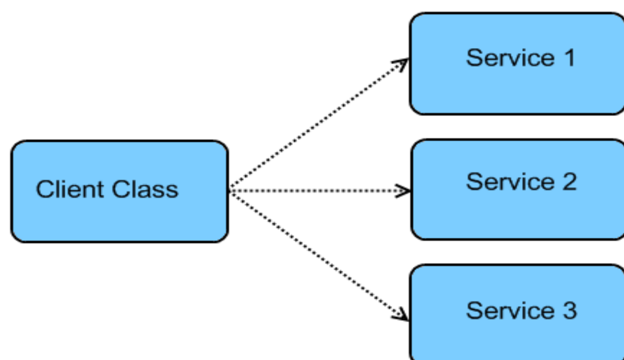
Microsoft provides wonderful tools that let a developer quickly deliver applications but in most cases there is no adherence to good software architecture principles such as Model-View-Control and design patterns as most of the logic and data access code are in the code behind of either the ASP.NET or Silverlight page. This is fine for many applications if they have a short half-life. However if you are developing a solution with a more long term view then one should look at employing good Object Orientated (OO) principles on the server side and hopefully the client side as well. Client side is difficult if you are using Javascript and that is why Silverlight is a great alternative solution.

The extra effort in the architecture will pay dividends in the long term as new features are added and to address software defects that might arise. The upfront development time turns out to be a small part of the effort in the life time of a product. Some of the patterns that I consider important are the above mentioned MVC and in addition its closely cousin Model-View-View-Model (MVVM). These should be employed along with the design patterns laid out by the Gang of Four in the book **Design Patterns: Elements of Reusable Object-Oriented Software.** For a much easier read, the **Heads First Design Patterns**, is available at http://oreilly.com/catalog/9780596007126. These are Java books, but at this level there is little difference to C# and if you really care there is real good C# Design Pattern web site at http://www.dofactory.com/Patterns/Patterns.aspx that can be of help.These can help deliver robust N-tier applications that will be around for years to come and help developers waste cycles on dealing with the spaghetti solution that result from not following these principles. This article will describe the development of a data gathering application using Silverlight 3 and Prism that will demonstrate using these practices.

Prism, or Composite Application Library (CAL) is set of guidance, conventions and code that is available from the MS Patterns. Prism is *an exercise in* object orientated design patterns and you should be familiar with these to get the full value of using Prism Instead of describing the details of all the patterns that are used in Prism I point you to http://www.silverlightshow.net/items/Patterns-and-practices-in-the-Composite-Application-Library-part-1.aspx . As I proceed thru the article I will highlight the pattern usage and only list them here:

- Composite User Interface patterns
  - Composite
  - Composite View
  - Command
  - Adapter
- Modularity patterns
  - Separated interface and Plug In
  - Service Locator
  - Dependency Injection
  - Event Aggregator
  - Façade
  - Registry
- Testability patterns
  - Inversion of control
  - Separated presentation
  -

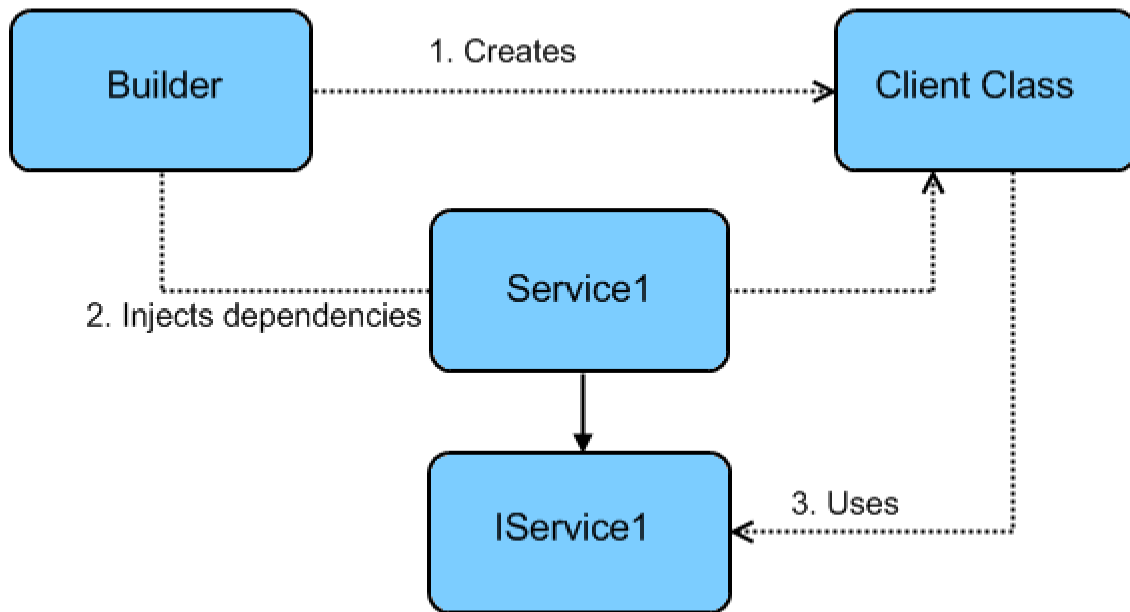
However I feel that is important to dive more into **Inversion of Control (IOC)** , **Dependency Injection (DI)** and **Modularity**. Here is the problem that we are trying to address:



With this architecture the client has a direct dependency on components whose concrete type is specified at design time and this leads to following problems:

- Your class is tightly coupled.
- In order to update the dependencies, you have to change your class code.
- The concrete implementations of the dependencies must be available at compile time.
- Your class contains code for creating, locating, and managing its dependencies, which is not possible to be reused.
- Your **ClientClass** is impossible to be tested, because it has a direct reference to its dependencies, which cannot be replaced with stubs or mocks.

This is the problem that DI and IOC address and these are well described by Martin Fowler at http://www.martinfowler.com/articles/injection.html but I want to add that the term IOC (Inversion of Control) is little heavy and as Martin points out misleading. Martin describes an architecture built around plug-ins or components that hide their implementation behind interfaces and these must be assembled into an applications (**loosely coupled classes).** We want to delegate the functions of selecting a concrete implementation type for the dependency to an **external component** or container. This is what IOC is all about but as Martin says IOC is too generic a term that people find confusing and DI is a more specific name for IOC whose goal is to reduce the dependency to one, the container that Prism provides when it is used. **Service locator** pattern is another variety of IOC that Prism uses and it allows classes to locate components they need without knowing the implementation. Now we have a decoupled, built for change architecture that looks like:



Modularity is an important attribute of Prism and is designing a system as a set of functional units or modules. Each module is an independent unit that can contain different components such as views and logic. Now we want to be able to locate and load modules at runtime in a **loosely coupled way** and Prism provides this key architecture thru **IModule** that allows us to follow Martin Fowler's Separated Interface. This simply means that the interface definition and implementation are in different assemblies so the client programs to the interface not the implantation. This is an idea that has been around for years as COM and DCOM then Web Services are built around it. Don't get me wrong, you can get started using prism without in-depth knowledge of the above but to get full use of Prism it will help.

Originally I looked at using ASP.NET MVC framework for the project and extensively use Javascript and JQuery for refining the user experience. I really like the MVC framework on the backend as it enforces using good design patterns. I decided to switch to Silverlight when I became aware of the Prism framework. Silverlight on its own is a compelling platform as it offers a rich development platform where the developer can use the same skills they do in delivering server side solutions. Thus we are able to use the Object Orientated methods with type

safety that is hard to replicate in Javascript. Now these are strong arguments on their own but then we thrown in Prism and we have a great patterns based development platform. We can now deliver agile, loosely coupled solutions that will allow the resultant applications to be easily changed and enhanced over time. Furthermore the framework lends itself to unit testing as we can easily inject mock implementations that contain the tests. I will not dive into details on Prism and the Unity framework as there are good articles on the Internet such as http://ira.me.uk/2009/03/06/building-a-composite-wpf-and-silverlight-application-with-prism-part-3/

The data access uses Ria Data services in the first version and later we will look at using Idea Blade's Devforce. The database itself is hosted in Microsoft new SQL Azure. The RIA Data services were chosen as an alternative to using basic WCF services as it simplifies writing a N-Tier application, its change tracking makes it ideal for writing an app like this. Again I will just point the reader to other articles and a video at: http://www.sandkeysoftware.com/Silverlight/ArticleNavigator/ArticleNavigator.Web/ArticleNavigatorTestPage.html

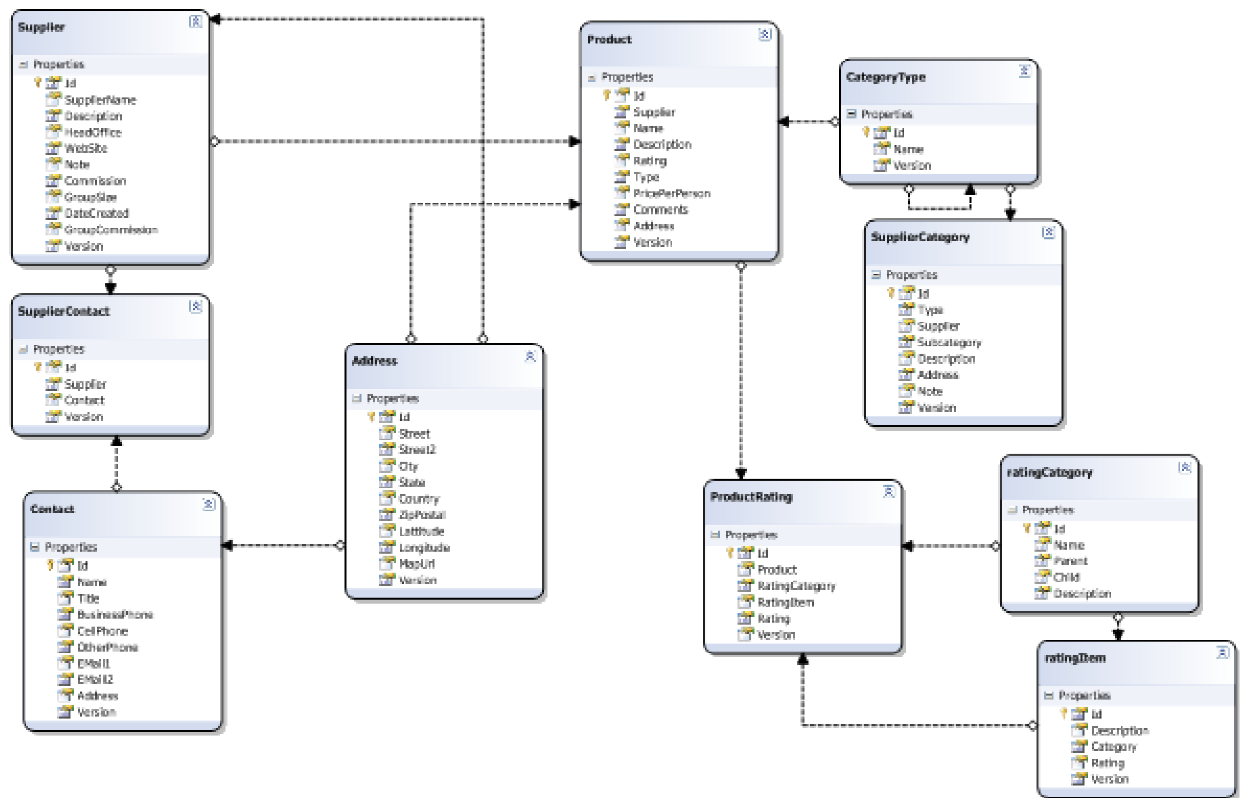
To follow along with this article there are a few downloads that are required:

1- RIA data services that it should be noted are still in a pre release form but very usable. They can be loaded from http://www.microsoft.com/downloads/details.aspx?FamilyID=76BB3A07-3846-4564-B0C3-27972BCAABCE&displaylang=en
2- Prism and the Unity Framework that can be loaded from http://www.microsoft.com/downloads/details.aspx?FamilyID=fa07e1ce-ca3f-4b9b-a21b-e3fa10d013dd&DisplayLang=en#Instructions

After performing the above the first step is to open up a standard Silverlight application, and make sure that you click on the enable RIA services button. This will allow us to blend the Silverlight project WizardRIA with the middle tier that is in the WizardRIA.Web project. I am using a custom database called Green that

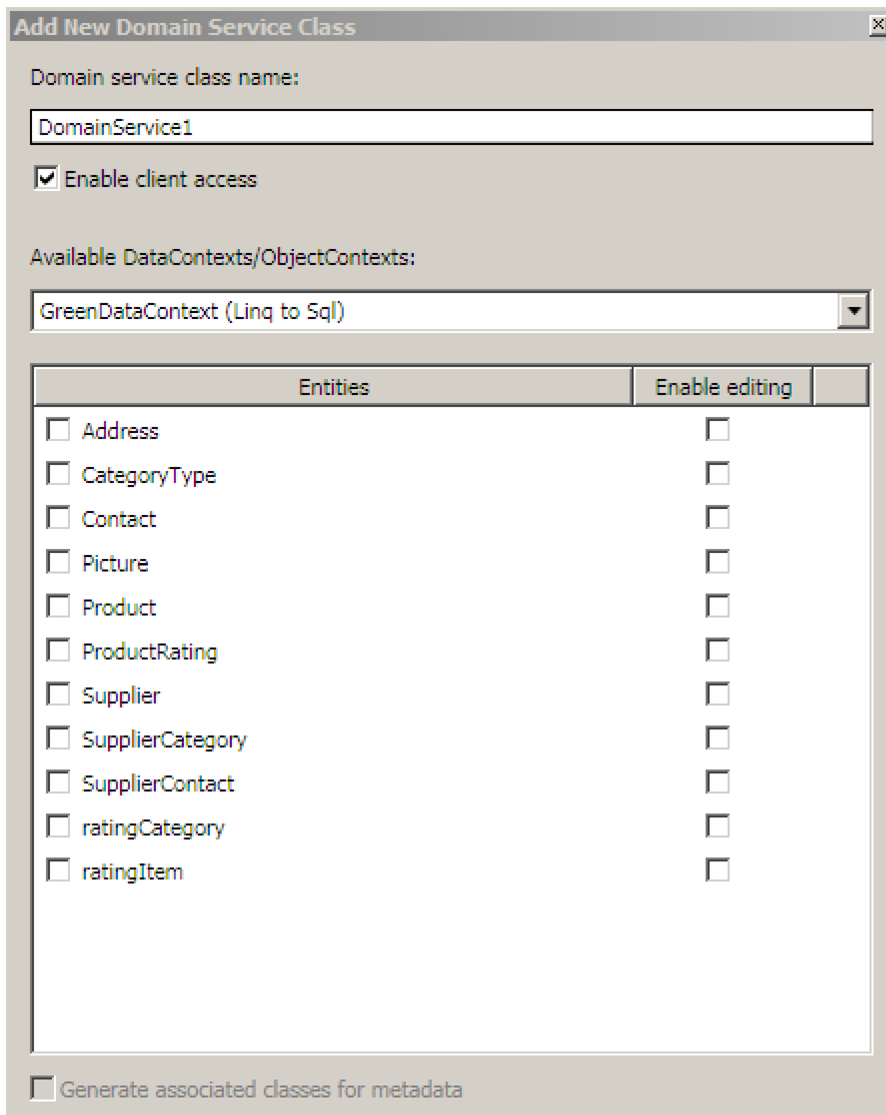
will allow us to define suppliers of resources such as motels and hotels. These resources have associated list of contacts. There are two major relationships that are defined:

1. Supplier is the vendor that is providing one or more resources such as the motel and these are referred to as products. One of the relationships is a Join table between the Supplier and Contacts so there is no direct binding and Contacts could be joined to other tables.
2. Each product can have ratings that are based on criteria defined by the UN. These ratings are defined in the RatingCategory and its associated RatingItem tables. There is a parent-child relationship here as a Category contains a list of child items.



I will assume that the reader is familiar with Linq and Linq to SQL as the next thing we do is to drop the tables from the Green data base on the Linq to SQL design surface. I said we will be moving fast we then click on Add new Item, Domain Service Class. The RIA services will define the client side classes that we will program against and the change control that will ease the persistence

code. The idea is that we are spreading the definition of the class across the client boundary .We are then presented with a dialog that lets use chose the tables to add to Domain source as follows:



It is tempting to grab all the Entities and add them to the Domain source at once but I have found that the best solution is to add the Entities one by one using partial classes. Then it is easier to make individual changes to the entities as an example I have defined a starting Domain class called GreenDomainService that contains only the Supplier domain, its definition look like:

```
[EnableClientAccess()]
```

```
Public partial class GreenDomainService :
LinqToSqlDomainService<GreenDataContext>
```

So now we can spread the definition of the DomainService across many files as in the following where we add the ProductDomainService:

```
public partial class GreenDomainService :
LinqToSqlDomainService<GreenDataContext>
   {
        public IQueryable<Product> GetProducts()
       {

       }
```

Note: when we use this approachVisual Studio will generate an [EnableClientAccess()] which we must remove as there can only be one of these for a project as it is what the framework keys on to define the client side version of the domain classes.

Following this process I ended up with the following .cs files :

AddressDomainService.cs

CategoryTypeDomainService.cs

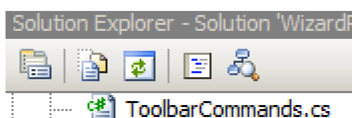
ContactDomainService.cs

ProductRatingsDomainService.cs

RatingsDomainService.cs

ProductDomainService1.cs

These also have their associated metadata.cs where we will can add attributes for things like validation. Now when we build the project the client side versions of these files are generated in the WizardRIA.Web.g.cs. This will only appear if we click on the Show All Files under the Solution Explorer:

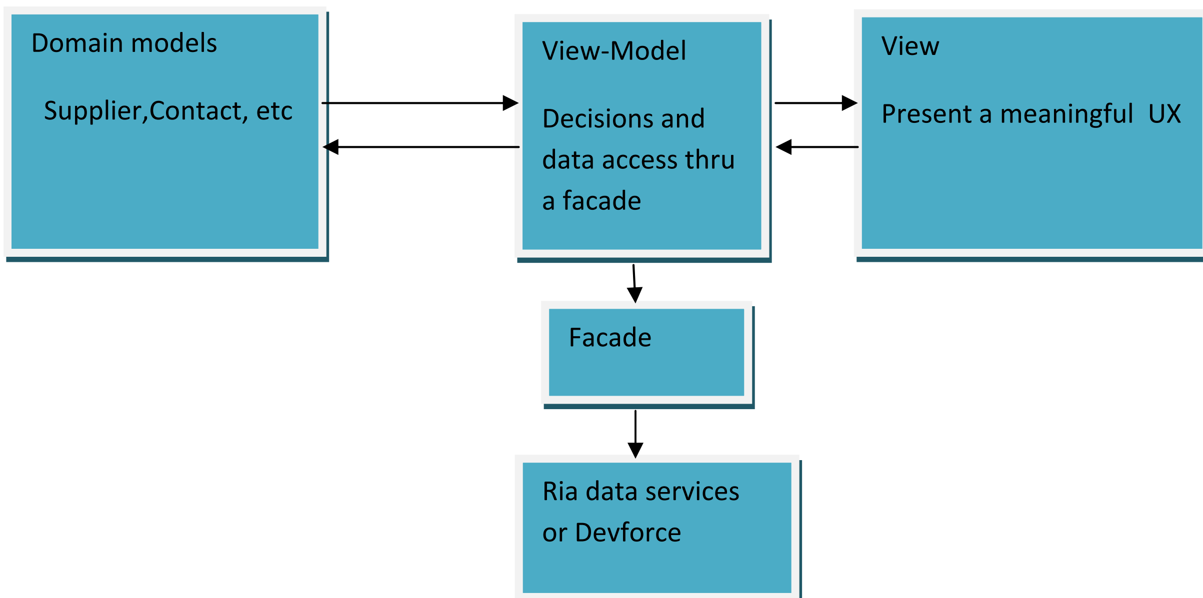


The generated classes that we will refer to as Models can be used to perform a subset of Linq to SQL operations from our Silverlight classes using asynchronous operations in code like this that will retrieve a list of all suppliers:

```
LoadOperation loadop = dc.Load<Supplier>(dc.GetSuppliersQuery());
loadop.Completed += new EventHandler(loadop_Completed);

void loadop_Completed(object sender, EventArgs e)
{
        throw new NotImplementedException();
}
```

Note:dc.Suppliers will contain the a list of Supplier entitys. These entities are the client side versions of the Domain classes and contain the attributes for validation and identity management.

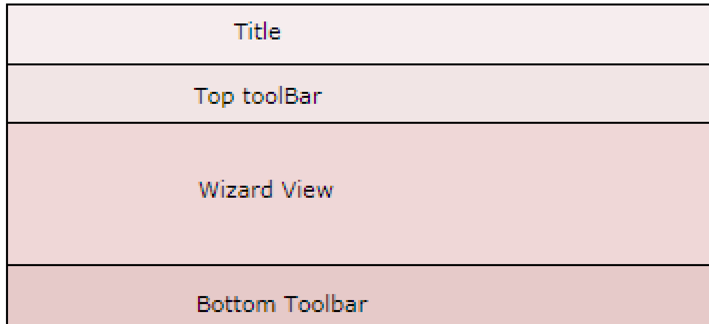
If this was a typical Silverlight application we would be pretty well done but then we would be using code in the code-behind file of the XAML file and be tying our controller code directly to the view. This is fine for small one of kind applications but it fragile if are building a robust web application like I would like my Wizard application to be. So we want to evolve to a more agile archtiecture where there is loose binding between the data access and our views. I had previouly discussed the Model-View-View-Model(MVVM) pattern in my video on my silverlight site mentioned above, in that article I show how MVVM can be used with my own dependency injection to achieve some of the decoupling that we are looking for. For a quick summary here is the idea, the View is only resposible for the visuals of the user experience and contains NO logic or data access. It defers these to the Viewmodel which ideally has no knowledge of the view.



However I then was intoduced to Prism and the Unity framework which I really am impressed with as it formalizes these processes. But now we have make architectural changes to our project

1. **DataServices**. This will contain an interface called IDataServices which will provide a façade for our data operations needed for the Wizard. The operations are implemented in a class called RIADataServices.cs
2. **Models** which will contain the above mentioned **WizardRIA.Web.g.cs**. I manually moved the file from where the tool place the file an operation that I do not like and hope MS provides an option in the code generator process like Devforce does.
3. Modules are the basic unit of prism and are what we associate with the views thru runtime dependency injection. We have a module project for following:
    a. Supplier
    b. Contacts
    c. Products
    d. Ratings
    e. TopToolbar
    f. Bottomtoolbar
4. InfraStructure contains utilities that are used across the modules

To use Prism we must change the startup process in our WizardRIA project which is the main Silverlight project. **Shell.XAML** is added as it acts as the visual host using the **Composite View** pattern and contains no logic. We will divide the main view into 4 Views that will look like:




```
<UserControl x:Class="WizardRIA.Shell"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     xmlns:Regions="clr-
namespace:Microsoft.Practices.Composite.Presentation.Regions;assembly=Micros
oft.Practices.Composite.Presentation"
    >
    <Grid x:Name="LayoutRoot" Background="White">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="50"></RowDefinition>          Title
                <RowDefinition Height="50"></RowDefinition>        TopToolbar
                <RowDefinition Height="*"></RowDefinition>          current  view
                <RowDefinition Height="50"></RowDefinition>          bottomtoolbar
            </Grid.RowDefinitions>
            <Grid   Background="#FF094B0D">
            <TextBox  Text="Green feet Travel - Choices for people who travel softly"
HorizontalAlignment="Center" Background="#FF094B0D" FontSize="25"
Foreground="LightGreen">

           </TextBox>
                </Grid>
             <!-- NOTE the use of HorizontalContentAlignment so that the content
stretchs to fill its container width-->
  <ContentControl  Grid.Row="1"
Regions:RegionManager.RegionName="Toolbar"
HorizontalContentAlignment="Stretch"></ContentControl>
            <ContentControl  Grid.Row="2"
Regions:RegionManager.RegionName="Form"
HorizontalContentAlignment="Stretch"
VerticalAlignment="Stretch"></ContentControl>
```

```
            <ContentControl  Grid.Row="3"
Regions:RegionManager.RegionName="BottomToolbar"
HorizontalContentAlignment="Stretch"></ContentControl>
        </Grid>
    </Grid>
</UserControl>
```

It is the RegionManager that will perform the work here as there is no code in the associated code-behind file. It will load the associated modules and their dependencies into their corresponding views.

The wizard view will contain the views for Supplier, Contacts and Products where we will collect the user input for these entities. The last view will allow the user to rate the product using a Treeview and GridView from Telerik. The nice thing agout using Prism is that I only have to define the Title view and the toolbar once as they do not change and fire the user button clicks to the viewModel that is active in the wizard view thru Prism **Commanding**.

The actual startup code is in BootStrapper.cs which will create the shell, configure the modules which will project a UI that appears in the regions defined by the Shell. It contains the following code:

```
public class Bootstrapper : UnityBootstrapper
    {
        protected override DependencyObject CreateShell()
        {
            Shell shell = this.Container.Resolve<Shell>();
            Application.Current.RootVisual = shell;
            return shell;
        }
 protected override Microsoft.Practices.Composite.Modularity.IModuleCatalog
GetModuleCatalog()
        {
            ModuleCatalog catalog = new ModuleCatalog();
            catalog.AddModule(typeof(SupplierModule));
            catalog.AddModule(typeof(ContactsModule));
            catalog.AddModule(typeof(ProductsModule));
            catalog.AddModule(typeof(RatingsModule));
            catalog.AddModule(typeof(ToolbarModule));
```

This the unity container as class derives from UnityBootstrapper so we introduce its functionality

Modules are loaded at runtime and associated with a View. These can be delay loaded as they are needed

```
        catalog.AddModule(typeof(BottomModule));

        return catalog;

    }
```

Now we need to change the app.xaml.cs

```
private void Application_Startup(object sender, StartupEventArgs e)
{
        new Bootstrapper().Run();
}
```

The call to Run will cause the above CreateShell code to be called and the UI to be displayed

Now we did a lot of setup and really do not have much to show but the real value is in the UnityBootstrapper which brings in the prism services such as Module loading and Dependancy Injection.

TopToolBar

TopToolbar contains the XAML for buttons like Add, Edit , Back and the relevant Next (Contacts, products or Rating). However it also performs all the data operations that we need for the views. The idea is that the data is retrieved up front and passed thru the Message to the active view that will use the contained object graph to add new objects or changes. Ria Data services will track all changes for us so when we reach the end of the wizard the object graph will reflect the user input and then can be persisted by simply calling the Save method of our DataAccess class. In the case of the Ria implementation we only have to call SubmitChanges .

As mentioned above Prism requires us to define a Module for the view and in this case we register the IDataServices dependancy and its implementation class RiaDataServices. We also register the ToolbarView with the Toolbar region as its only view.

```
public class ToolbarModule : IModule
```

```
    {
        private IUnityContainer container;
        private IRegionManager regionManager;
        public ToolbarModule(IRegionManager regionManager, IUnityContainer
container)
        {
            this.container = container;
            this.regionManager = regionManager;
        }
        public void Initialize()
        {
            this.container.RegisterType<IDataServices, RIADataServices>(new
ContainerControlledLifetimeManager());
            this.regionManager.RegisterViewWithRegion("Toolbar",
typeof(ToobarView));
        }
    }
```

Here is where the Unity DI container kicks in

Now what about this DI stuff, how does that work, well lets look at the definition (at least part of it) of the ToolbarView:

```
public partial class ToobarView : UserControl
    {

        private IEventAggregator eventAggregator = null;
         private TopViewModel vm = null;
         public ToobarView(TopViewModel viewmodel, IEventAggregator
eventAggregator)
        {
            InitializeComponent();
            this.eventAggregator = eventAggregator;
            LayoutRoot.DataContext = vm = viewmodel;
             //the ViewModel will call us to change bitton state
            eventAggregator.GetEvent<SetViewButtons>().Subscribe(ChangeButtons);

        }
```

Unity DI will use reflection to see that it needs to create the VM and event aggregator

Databinding will use public properties of the VM

The ViewModel has a constructor that looks like:

```
 public TopViewModel(IEventAggregator eventAggregator, IDataServices ds)
        {
```

```
message = new EntityNotificationMessage();
DataAccess = ds;
DataAccess.GetSuppliers(OnGetSuppliersComplete);
this.eventAggregator = eventAggregator;
```

the line of code in ToolbarModule.cs :

```
this.regionManager.RegisterViewWithRegion("Toolbar", typeof(ToobarView)
```

will kick in the Unity DI Container that will first ask the question, what is ToolBarView and using reflection will find the above constructor in the View and notice that it must construct a ViewModel and eventAggregator. This is refered to as Constructor injection. As a result the View and VM are injected by the module initializer along with the RiaDataAccess class.

The other key is that prism makes the use of MVVM rather natural as we only had to add one line of code to the Views constructor to set Datacontext to our Viewmodel:

```
LayoutRoot.DataContext = vm = viewmodel;
```

Notice we are also passing in IEventAggregator interface will allow use to use Commanding in this view. We then use it to subscribe to the SetViewButtons from the ViewModel as it is key to remember that the view is completely decoupled from the ViewModel. We could have given the ViewModel knowledge of the view to make this less complex but then we are adding dependancys .

We now have to address how we fire events from the View's XAML back to the ViewModel for example as the user clicks on the Add button. This is where Prism's Commanding comes in as there nothing in Silverlight like WPF's commanding.Commanding provides a loosely coupled way to bind the UI to the logic that outside the logical tree and do not require handling in the code behind. We first define a property in TopViewModel for the AddCommand that looks like:

```
public ICommand AddCommand { get; private set; }
```

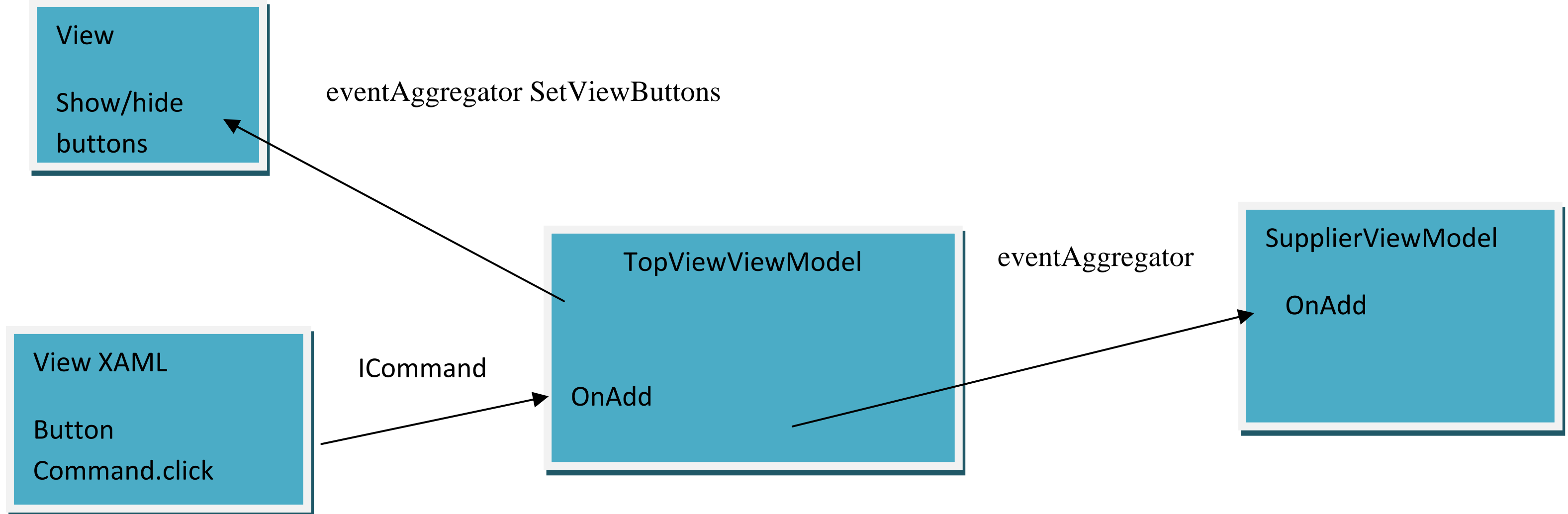
We then use the DelegateCommand as follows:

```
AddCommand = new DelegateCommand<string>(OnAdd);
```

This will allow us to add code to our XAML that will result in the OnAdd action being called. Then we modify our button XAML to look like:

```
<Button x:Name="Add" Content="Add"  Grid.Column="3"
Style="{StaticResource GlassButton}"   Commands:Click.Command="{Binding
AddCommand }" ></Button>
```

EventAggregation is also how we bidirectional communicate between the active view and the Toolbars. We use Prism Commanding between buttons of XAML and VM so the communication paths look like:

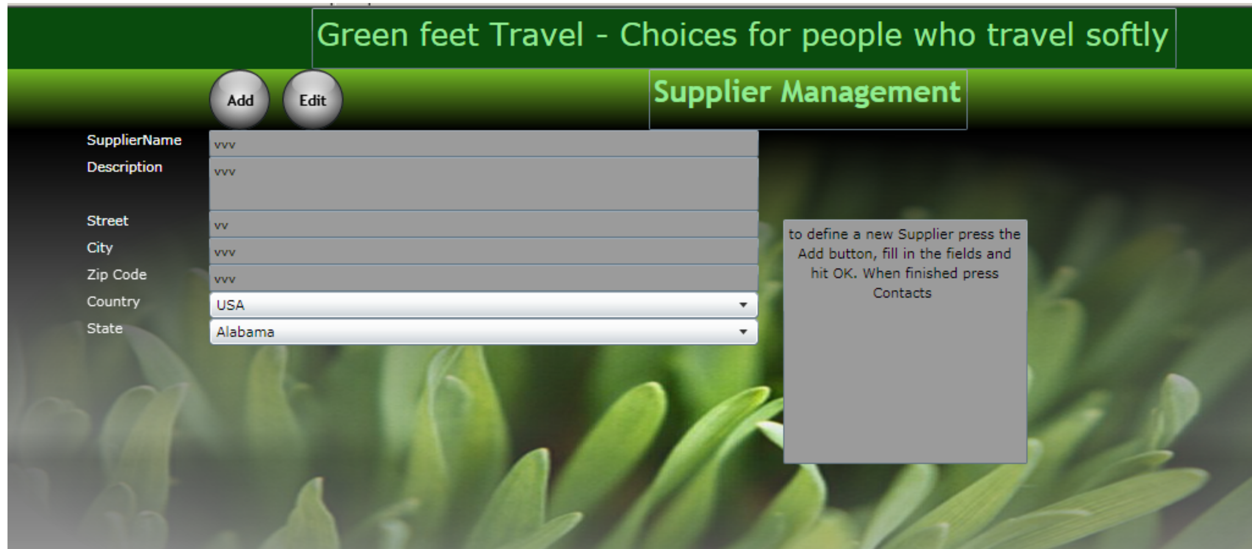


Data Access

All data is being retrieved from the middle tier on the server thru the IDataAccess interface which is instantiated as the RIADataServices implementation by Unity . The data access is performed asynchronously in the ViewModel as this is Silverlight and placed in the Message object that will be passed to the active view. When the last entities have been retrieved we then set the active view to Supplier then send SetViewButtons command to the view to enable the buttons for the state the active entities. For example intially there is only an Add button displayed. The Open command is then fired to the Supplier view so it will activate its view.

Wizard Views

We now look at the four views that will make up our wizard as only one of them is active at one time and the user naviagates forwards or back with the buttons. I will only go into details on the SupplierInfo that presents a user with a collection dialog that looks like:



I wanted to demonstrate an alternative to the Constructor injection that I used in the TopToolBar and adopted a Factory pattern based on a wonderful article by Ward Bell of IdeaBlade at

http://www.ideablade.com/DevforceSilverlight/DevForceSilverlight_PrismExplorer.aspx

As such I make a change in the Initialize method to resolve SupplierFactory

```
using Microsoft.Practices.Composite.Modularity;
using Microsoft.Practices.Composite.Regions;
using Microsoft.Practices.Unity;
using DataServices;

public class SupplierModule:IModule
   {
```

```
#region IModule Members
private IUnityContainer container;
public SupplierModule(IUnityContainer container)
{
   this.container = container;
}
public void Initialize()
{
   container.Resolve<SupplierFactory>();
}
```

We create a basefactory class that will use **Service locator** instead of **Constructor injection**. Supplierfactory extends this and uses the Subscribe method to insure that the view will only be activated when an Open command is received, this was fired from the TopToolbar as the result of a forward or back navigation. The SupplierFactory looks like:

Note the use of the Service locator

```
public class SupplierFactory : BaseFactory
{
    public SupplierFactory(IServiceLocator locator, IRegionManager
regionManager, IEventAggregator eventAggregator):base( locator,
regionManager,  eventAggregator)
    {
       handler = OnOpen;
       IsRevelant = message => null != message &&
message.IsEntityType<Supplier>();
       Subscribe();
    }
     public void OnOpen(EntityNotificationMessage Message)
    {
       CreateView(typeof(SupplierViewModel),typeof(SupplierView));
       (_moduleViewModel as SupplierViewModel).OnOpen(Message);
     }
```

Lambda insures Open is only called if Supplier is active view

Call open for supplier that will databind Active object properties to view from VM

The base class is where most of work is done and looks like:

```
public class BaseFactory
  {
     protected  IServiceLocator _locator;
     protected  IRegionManager _mainRegionManager;
```

```
        protected  IEventAggregator _eventAggregator;
        protected FrameworkElement _moduleView;
        protected object _moduleViewModel;
        protected EntityNotificationMessage _Message;

        public BaseFactory(IServiceLocator locator, IRegionManager
regionManager, IEventAggregator eventAggregator)
        {

            _locator = locator;
            _mainRegionManager = regionManager;
            _eventAggregator = eventAggregator;

        }
        protected Action<EntityNotificationMessage> handler = null;
        protected Predicate<EntityNotificationMessage> IsRevelant = null;

        protected void Subscribe()
        {

            _eventAggregator.GetEvent<OpenCommand>().Subscribe(
            handler,      // the function that implements the response
            ThreadOption.UIThread, // invoke subscription on the UI Thread
            true,                  // true KeepAlive means ensure strong fn references
            IsRevelant // Fn to filter events so only hear pertine
            );
        }
        /// </summary>
        /// <param name="mv"></param>
        /// <param name="view"></param>
        protected void CreateView(Type mv,Type view)
        {
            _moduleViewModel = _locator.GetInstance(mv);
            _moduleView = _locator.GetInstance(view) as FrameworkElement;
            _moduleView.DataContext = _moduleViewModel;

           var exists =
_mainRegionManager.Regions["Form"].Views.Where(o=>o.GetType() ==
view).SingleOrDefault();
            if (exists != null)
```

Delegates that define the method (handler) to be called if the IsRevelant predicate is true

Subscribe to the OpenCommand

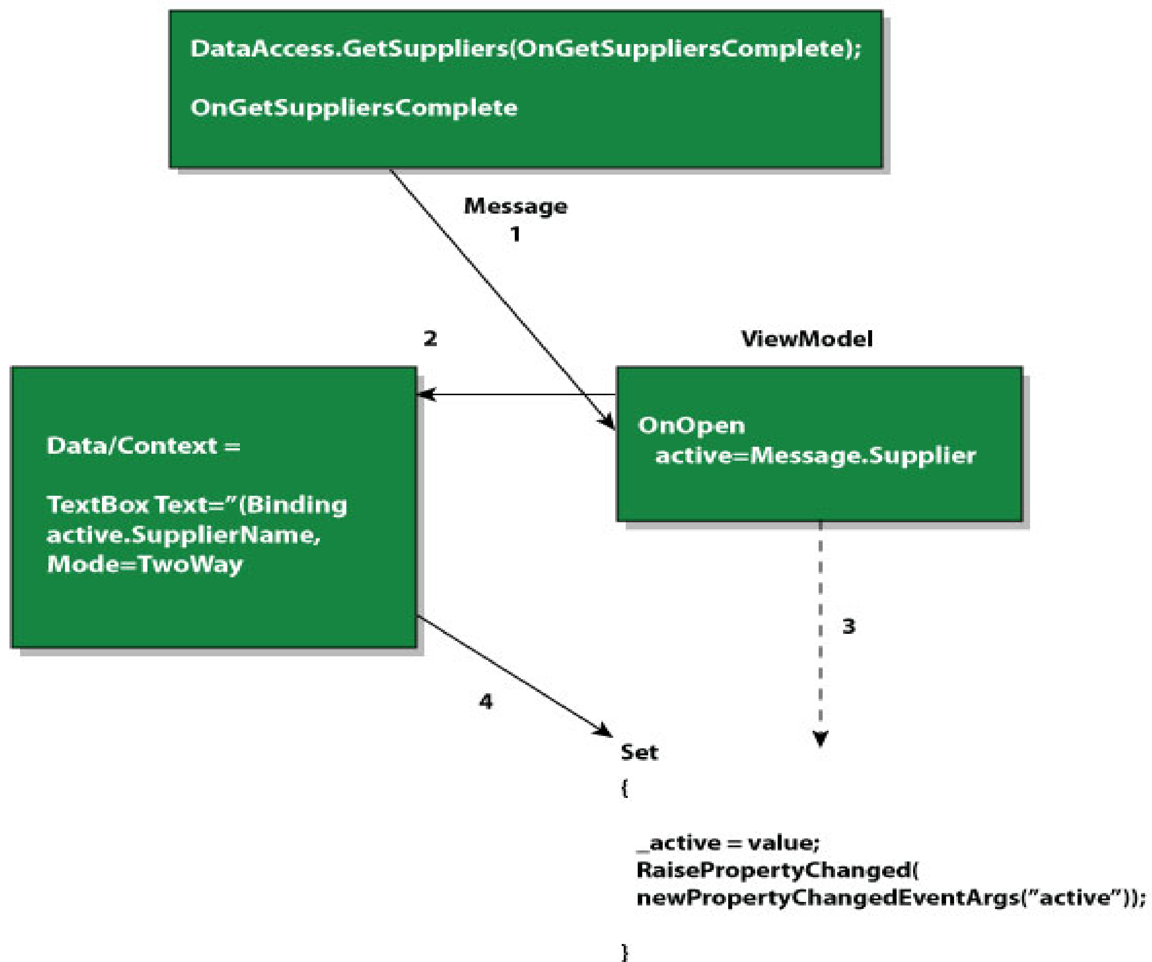
CreateView will do the heavy lifting for the Prism stuff as we use locator services to get new instances of the ViewModel and View

If the view already exists then remove it

```
            _mainRegionManager.Regions["Form"].Remove(exists);
          _mainRegionManager.AddToRegion(RegionNames.MainRegion,
_moduleView);
          _mainRegionManager.Regions["Form"].Activate(_moduleView);

        }

    }
```

ViewModel

This data is passed to the ViewModel as the result of an Open Command being called by the SupplierFactory .As a result the view is updated from the Viewmodel using databinding. This is automatic process as all the input entiies derive from INotifyProperty and must be because there is no way of knowing when the data is available. A good knowledge of Silverlight databinding is a requist so please check out my article at my Silverlight site mentioned above .



1 GetSuppliers is called which results in all Supplier and related entities being queried from middle tier. When this factory receives OnOpen, the VM and view are created and OnOpen is called in the instantiated ViewModel.

2 the View constructor binds the ViewModel to the View so we can databind against the ViewModel active property.

3 OnOpen is received and it does active=message.Supplier[0] , Setter is called for active that results in **RaisePropertyChanged** called for active object as our ViewModel extends INotifyPropertyChanged.

4 there is a "Listener" thru the two-way mode and changes are made to UI

```
public class SupplierViewModel : ViewModelBase
    {
        int current = 0;
        public Supplier _active;
        IEventAggregator eventAggregator = null;
        public SupplierViewModel()
        {

        }
         public Supplier active
        {
            get { return _active; }
            set
            {
                _active = value;
                 RaisePropertyChanged(new PropertyChangedEventArgs("active"));
            }
        }
       public SupplierViewModel(IEventAggregator eventAgg)
       {
          eventAggregator = eventAgg;

         eventAggregator.GetEvent<AddCommand>().Subscribe(OnAdd,ThreadOpti
         on.UIThread,   true,   IsNotificationRelevant );

         eventAggregator.GetEvent<OKCommand>().Subscribe(OnOK,ThreadOptio
         n.UIThread,   true,   IsNotificationRelevant);
          eventAggregator.GetEvent<EditCommand>().Subscribe(OnEdit);
```

property for the active supplier that contains the RaisePropertyChanged that will allow us to bind to XAML thru DataContext

Subscribe to Add,Edit and OK Commands from the toolbars

```
        }

        public static bool IsNotificationRelevant(EntityNotificationMessage message)
        {
            return null != message && message.IsEntityType<Supplier>();
        }

        public void OnOpen(EntityNotificationMessage Message)
        {

            if (Message.Suppliers.Count == 0)
                active = new Supplier();
            else
                active = Message.Suppliers[current];
        }

        public void OnAdd(EntityNotificationMessage m)
        {


            active = new Supplier();
            active.Address = new Address();
            eventAggregator.GetEvent<AddViewCommand>().Publish("g");
            bAdding = true;

        }
```

make sure we only receive the above commands when we are the

Active setter starts the data binding process

Only my code knows my object hierarchy so cannot use dataform as I create embedded objects

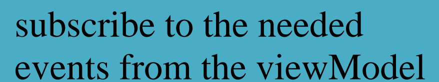
Send an addview command to Supplierview to update its state

Open will leave the input form in a readonly state so when the user presses the Add button that is processed by the TopToolbar button , a AddViewCommand command must be fired from the ViewModel to the view. This will result in user input being enabled in the view.Now we could have given the ViewModel knowledge of the view but that would be breaking the decoupling that we are looking for. Of course Add will create new Supplier and its child entity address as the form will populate these fields. I originally hoped to use the Siliverlight control called Dataform as it has much of the function that I needed but I could not

find a way to supply it with the indepth knowledge of my object graph needed for this step. Like most of the supplied controls it is good only for simple object graphs.

SupplierView

```
public partial class SupplierView : Viewbase
   {

       private SupplierViewModel vm
       {
          get { return DataContext as SupplierViewModel; }
       }

       public SupplierView(IEventAggregator eventAgg)
       {
          InitializeComponent();
          eventAggregator = eventAgg;

        eventAggregator.GetEvent<EditViewCommand>().Subscribe(OnEdit);
          eventAggregator.GetEvent<AddViewCommand>().Subscribe(OnAdd);
        eventAgg.GetEvent<AddShowErrorCommand>().Subscribe(OnError);

          //set the background bitmap
          backBmp.Source = bmp;
          //set an initial value as this can be changed by country selection change
          ZipPostal.Text = "Zip Code";
          //set the textboxes to readonly and the column definitions
          Utilities.GetTextBoxes(this.LayoutRoot, textBoxList);
          textBoxList.ForEach(t => { t.Background = new
SolidColorBrush(Colors.Gray); t.IsReadOnly = true; });
          setRowColumndefinitions(MyForm);

       }
```



The code in the view that processes this AddViewCommand command fired from the ViewModel is :

```
public void OnAdd(string title)
       {
          Utilities.GetTextBoxes(this.LayoutRoot, textBoxList);
```

```
            textBoxList.ForEach(t => { t.Background = new
SolidColorBrush(Colors.LightGray); t.IsReadOnly = false; });

        }
```

There is additional code in the View and its base class that just deals with UX issues like populating the dropdowns for the States and provinces. For example when the dropdown for country changes we change the Itemsource of StatesProvs to either Canadian provices or states as in :

```
cbStatesProvs.ItemsSource = vm.ProvInfo;
public void OnOpen(EntityNotificationMessage Message)
```

DataServices

The implementation of the IDataServices interface is contained here and use Ria Data services to implement the following methods:

```
public interface IDataServices
    {
        void GetSuppliers(Action<EntityList<Supplier>> callback);
        void GetCategoryTypes(Action<EntityList<CategoryType>> callback);
        void Save(Action<string> callback);

        void GetRatingInfo(Action<EntityList<ratingCategory>> callback);

    }
```

All of these method definitions take an Action that returns an EntityList . the usage pattern starts with the implementation that looks like:

```
GreenDomainContext dc = new GreenDomainContext();
        public void GetSuppliers(Action<EntityList<Supplier>> callback)
        {
```

```
            LoadOperation loadop = dc.Load<Supplier>(dc.GetSuppliersQuery());
            //Completed is an eventhandler that takes an Action delegate so just use the
one we passed in as a parm
            loadop.Completed += (sender, e) => callback(dc.Suppliers);
        }
```

As a result when the Load operation is completed an Entity List of Suppliers is returned to the caller that issued the following call:

```
DataAccess.GetSuppliers(OnGetSuppliersComplete);

private void OnGetSuppliersComplete(EntityList<Supplier> SupplierInfo)
{
}
```

The GetSuppliersQuery method in the GreenDomainContext results into a server call being made the GetSuppliers method of the GreenDomainService. Now I wanted to return the whole object graph for Supplier in one call so I used the Linq DataLoadOptions to do this as follows:

```
public IQueryable<Supplier> GetSuppliers()
{

            var accountLoadOptions = new DataLoadOptions();
            accountLoadOptions.LoadWith<Supplier>(s => s.SupplierContacts);
            accountLoadOptions.LoadWith<Supplier>(s => s.Address);
            accountLoadOptions.LoadWith<SupplierContact>(sc => sc.Contact1);
            accountLoadOptions.LoadWith<Contact>(c => c.Address1);
            accountLoadOptions.LoadWith<Supplier>(s => s.Products);
            accountLoadOptions.LoadWith<Product>(p => p.Address1);
           accountLoadOptions.LoadWith<Product>(p => p.ProductRatings);
            accountLoadOptions.LoadWith<Product>(p => p.CategoryType);
            Context.LoadOptions = accountLoadOptions;
            return this.Context.Suppliers;

        }
```

Now this will not result in the graph being serialized to the client and todo this we have to add **Includes** in each of the Metadata files associated with the DomainService :

```
internal sealed class SupplierMetadata
{
```

```
[Include]
public Address Address;

[Include]
public EntitySet<Product> Products;
[Include]
public EntitySet<SupplierContact> SupplierContacts;
```

The generated query looks like :

```
SELECT [t0].[Id], [t0].[SupplierName], [t0].[Description], [t0].[HeadOffice],
[t0].[WebSite], [t0].[Note], [t0].[Commission], [t0].[GroupSize],
[t0].[DateCreated], [t0].[GroupCommission], [t0].[Version], [t2].[Id] AS [Id2],
[t2].[Supplier], [t2].[Contact], [t2].[Version] AS [Version2], [t3].[Id] AS [Id3],
[t3].[Name], [t3].[Title], [t3].[BusinessPhone], [t3].[CellPhone], [t3].[OtherPhone],
[t3].[EMail1], [t3].[EMaill2], [t3].[Address], [t3].[Version] AS [Version3],
[t4].[Id] AS [Id4], [t4].[Street], [t4].[Street2], [t4].[City], [t4].[State],
[t4].[Country], [t4].[ZipPostal], [t4].[Lattitude], [t4].[Longitude], [t4].[MapUrl],
[t4].[Version] AS [Version4], (
   SELECT COUNT(*)
   FROM [dbo].[SupplierContact] AS [t5]
   INNER JOIN ([dbo].[Contacts] AS [t6]
      INNER JOIN [dbo].[Address] AS [t7] ON [t7].[Id] = [t6].[Address]) ON
[t6].[Id] = [t5].[Contact]
   WHERE [t5].[Supplier] = [t0].[Id]
   ) AS [value], [t1].[Id] AS [Id5], [t1].[Street] AS [Street3], [t1].[Street2] AS
[Street22], [t1].[City] AS [City2], [t1].[State] AS [State2], [t1].[Country] AS
[Country2], [t1].[ZipPostal] AS [ZipPostal2], [t1].[Lattitude] AS [Lattitude2],
[t1].[Longitude] AS [Longitude2], [t1].[MapUrl] AS [MapUrl2], [t1].[Version] AS
[Version5]
FROM [dbo].[Supplier] AS [t0]
INNER JOIN [dbo].[Address] AS [t1] ON [t1].[Id] = [t0].[HeadOffice]
LEFT OUTER JOIN ([dbo].[SupplierContact] AS [t2]
   INNER JOIN ([dbo].[Contacts] AS [t3]
      INNER JOIN [dbo].[Address] AS [t4] ON [t4].[Id] = [t3].[Address]) ON
[t3].[Id] = [t2].[Contact]) ON [t2].[Supplier] = [t0].[Id]
ORDER BY [t0].[Id], [t1].[Id], [t2].[Id], [t3].[Id], [t4].[Id]
```