

Objects Only

In chapter 1 we looked at a simple RIA application and investigated some CRUD operations against a relational database using Linq To SQL. This chapter will explore an important aspect of the framework in that it can be used with pure object data sources so we can persist to things like N-Hibernate or XML files which we will explore in this chapter. The first thing we do is define XML to save our entities to; in this case the entity is a City class that looks like:

```
public partial class City
{
    [Key]
    public string Name { get; set; }
    public string State { get; set; }
}
```

Now we will use an XML file to persist and retrieve our objects from and the simple XML will look like:

```
<Cities>
  <City>
    <Name>Gainesville</Name>
    <State>Fl</State>
  </City>
  <City>
    <Name>Buffalo</Name>
    <State>NY</State>
  </City>
</Cities>
```

Unfortunately when we work with objects we have to do a little more typing as we must manually add the `CityService` which will derive directly from `DomainService`:

```
[EnableClientAccess()]
public class CityService : DomainService
{
    private CityData _cityData = new CityData();
    public IEnumerable<Models.City> GetCities()
    {
        return this._cityData.Cities;
    }
    public void UpdateCity(Models.City currentCity, Models.City
originalCity)
    {
        _cityData.ChangeCity(currentCity);
    }
}
```

include an update method so we can edit the entitys client side

Now we define methods in the the CityData class that will let us retrieve and update the objects to and from the XML backing file.

```
public class CityData
```

```
{  
    private static IEnumerable<City> _cities = null;  
    private static XElement xml = null;  
    public CityData()  
    {  
        xml = XElement.Load(HttpContext.Current.Server.MapPath("~/App_Data/Cities.xml"));
```

Load the XML from
Cities.xml at construction
time

```
    }  
    public IEnumerable<City> Cities
```

```
{  
    get  
    {  
        return LoadCities();
```

Use each city element in the
XML to hydrate a new City
object and add to list of cities

```
    }  
}  
private static List<City> LoadCities()  
{  
    List<City> cities = new List<City>();  
    foreach (XElement CityElement in xml.Elements("City"))  
    {  
        City c = new City();  
        {  
            c.Name = CityElement.Element("Name").Value;  
            c.State = CityElement.Element("State").Value;  
        }  
        cities.Add(c);  
    }  
    return cities;  
}
```

The user has made changes in the
client tier and they are reflected in
the input city object.

```
public void ChangeCity(City changed)
```

```
{  
    XElement myNode = null;  
    try  
    {  
        myNode = xml.Descendants("City").Where(x => x.Element("Name").Value ==  
changed.Name).SingleOrDefault();  
        if ( myNode != null)
```

Find the XML node in that has the
input city name and change the value
of the State

```

        myNode.Element("State").Value = changed.State;
    }
    catch (Exception e)
    {
        throw new ApplicationException("cannot find City node to change");
    }
    //persist the changes to the XML file
    string strFilePath = HttpContext.Current.Server.MapPath("~/App_Data/Cities.xml");
    xml.Save(strFilePath);
}
}
}

```

Client

Now most the work is done as the framework will generate the class that we need to perform our CRUD operations in the client tier. We just add CityContext to our Mainpage.cs and reference it in the constructor to load the Cities into our grid just as in our Simple example in Chapter 1. We can also change the dc.LoadCities to look like :

```

dc.LoadCities (CityContext.CitiesQuery.Where (c =>
c.State == "FL"), null);

```

As we also added an update method the the DomainService we can also edit the states in the dataGrid and when the save is clicked the changes are persisted to our XML file. We can also add the same validation attributes that we did in Chapter 1.