<http://www.codeproject.com/Articles/84908/WCF-RIA-Services-ASP-NET-Client-Web-Application>

N-Tier development with ASP.NET MVC, WCF, and LINQ

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | <http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ> |  | |  |

Top of Form

[**Download source code - 209 KB**](http://www.codeproject.com/KB/aspnet/aspnetmvc_bugtracker_v4/BugTracking.zip)

Introduction

This article is about a possible way to implement an **N-Tier** architecture for ASP.NET MVC (Model View Controller), using **WCF** (Windows Communication Foundation = replacement for .NET Remoting) and **LINQ to SQL**. This article will not go in depth explaining the ins and outs of MVC, WCF, and LINQ (there's already enough stuff which explains these technologies ...); instead, I'll be showing a possible way to implement these features in an N-Tier environment, by means of a simple bug-tracking demo application.

Requirements of the Bug-Tracking Demo Application

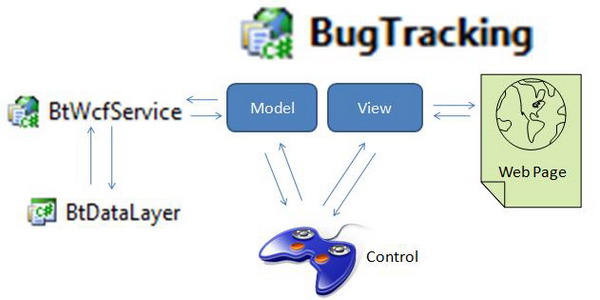
First, we should state the requirements of our demo application. These are quit straightforward, as mentioned beneath:

We want the application to produce a list of currently added bug-tracking tickets.

We want the possibility that a user can add a ticket to the system, which evolves adding a ticket date, selecting a user to whom we grant the ticket, adding a ticket date, and letting the user add some comments.

Before adding a new ticket to the system, we want to be sure that the user has supplied a valid date, selected a ticket type and user, and added some comments for the ticket.

Architectural Overview



The .NET Solution consists of three main projects:

**BtDataLayer**: Contains the model classes and the data access routines.

**BtWcfService**: Contains the services which the presentation layer may call.

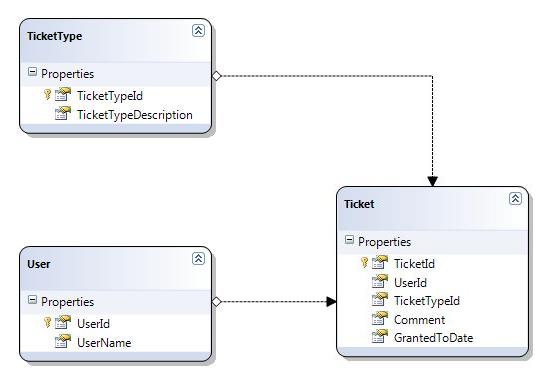
**BugTracking**: Contains the presentation layer (ASP.NET MVC app).

Another project not mentioned on the diagram above is **CommonLib**, which contains the commonly used routines (like a business base class which each model class derives from, method for cloning objects, and a business rule validator engine).

Using the Code

The Data Access Layer (BtDataLayer)

The model classes (*BugTracker.dml*):



These are the LINQ-TO-SQL classes retrieved from the SQL-Server Bugtracker database (database template included in the Zip file). As this model should be remotable, the serialization mode on the datacontext should be set to ***Unidirectional***.

The Data Access Code (*btDataManger.cs*):

We want to be able to load a user-list so we can select a user for whom the ticket applies to:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ)

public static List<user> GetUserList()

{

*// Create the DataContext to retrieve the Users from.*

BugTrackerDataContext db = new BugTrackerDataContext();

*// Return a list of Users.*

return db.Users.ToList<user>();

}

We want to be able to load a tickettype list, so we can select the type of ticket (BUG, INFO, etc ...) for the ticket:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ)

public static List<tickettype> GetTicketTypeList()

{

*// Create the DataContext to retrieve the Ticket Types from.*

BugTrackerDataContext db = new BugTrackerDataContext();

*// Return a list of TicketTypes.*

return db.TicketTypes.ToList<tickettype>();

}

We want to be able to retrieve all the current tickets from our ticket table:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ)

public static IEnumerable<ticket> GetTickets()

{

*// Create the DataContainer to retrieve the tickets.*

BugTrackerDataContext db = new BugTrackerDataContext();

*// Return a List of Tickets*

return db.Tickets;

}

Finally, we want to have the possibility to persist a newly added ticket (one at a time for the demo application, but we make the method all-purpose, so it can accept many added or modified objects):

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ)

public static bool PersistTickets(ref TicketList p\_tickets)

{

*// Only persist if any tickets to add or modify.*

if (p\_tickets == null || p\_tickets.Count == 0)

return false;

*// Create the persistence datacontext*

BugTrackerDataContext db = new BugTrackerDataContext();

*// Persist any new or modified tickets.*

foreach (Ticket t in p\_tickets)

{

if (t.TicketId == 0)

{

db.Tickets.InsertOnSubmit(t);

}

else

{

db.Tickets.Attach(t, t.IsDirty);

}

}

try

{

db.SubmitChanges(ConflictMode.ContinueOnConflict);

*// Reset the isDirty flag.*

foreach (Ticket t in p\_tickets)

{

t.IsDirty = false;

}

}

catch (ChangeConflictException ex)

{

throw ex;

}

return true;

}

The Service Layer

The purpose of our WCF enabled service layer is to make the model and data-access routines (to foresee the model of data ...) available to the presentation layer.

Our service layer consists mainly of two routines:

The service contract (which contains the routines which can be called by our Web App).

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ)

[ServiceContract]

public interface IBtService

{

[OperationContract()]

IEnumerable<user> GetUserList();

[OperationContract()]

IEnumerable<tickettype> GetTicketTypeList();

[OperationContract()]

bool PersistTickets(ref TicketList p\_tickets);

[OperationContract()]

IEnumerable<ticket> GetTickets();

}

The Service Implementation, which implements our contract and calls the datalayer.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ)

public IEnumerable<user> GetUserList()

{

return BtDataManager.GetUserList();

}

public IEnumerable<tickettype> GetTicketTypeList()

{

return BtDataManager.GetTicketTypeList();

}

public bool PersistTickets(ref TicketList p\_tickets)

{

return BtDataManager.PersistTickets(ref p\_tickets);

}

public IEnumerable<ticket> GetTickets()

{

return BtDataManager.GetTickets();

}

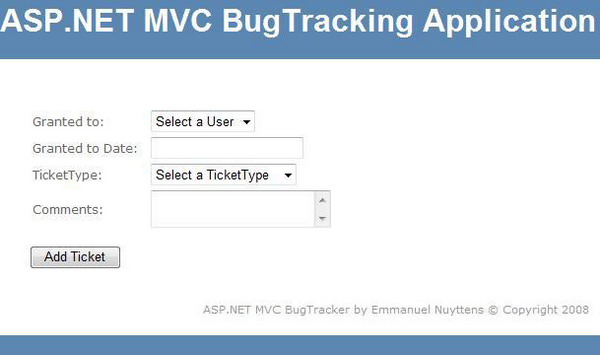
The Presentation Layer

The presentation layer contains our ASP MVC application. As already mentioned above, I will not go in depth on MVC, as there is already a lot of info available on the net. But it is important to know that MVC contains three main items, namely the **Model** (which represents our class, a validation logic, and is remotely available through the Service Layer), the **View** (our webpage), and the **Controller** (which basically "hooks" the Model to the View).

The Views and the Controllers

Our demo application hosts the following MVC enabled web pages:

***Ticket.aspx***



With this page, the user can add a bug-ticket to the system. The **HomeController** is responsible for rendering the page. While interacting with the page, there are two actions involved, one for creating a new ticket (the "GET") version, and one for persisting a new ticket (when the user hits the Submit button: the POST version).

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ)

[AcceptVerbs("GET")]

public ActionResult Ticket()

{

this.LoadSelectionLists();

*// Get the userlist*

var userList = new SelectList(\_userList, "UserId", "Username");

ViewData["UserId"] = userList;

*// Get the tickettypelist*

var ticketTypeList = new SelectList

(\_ticketTypeList, "TicketTypeId", "TicketTypeDescription");

ViewData["TicketTypeId"] = ticketTypeList;

return View();

}

When the user selects "Create Ticket" in the index page, we must create a new page where the user may enter the ticket details. As the user should have the possibility to select a user and ticket type for the ticket, these data are loaded from the service and added to the viewstate of the form:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ)

private void LoadSelectionLists()

{

*// Get a proxy to the data service provider*

BtServiceRef.BtServiceClient proxy = new

BugTracking.BtServiceRef.BtServiceClient();

*// Get the userlist*

\_userList = proxy.GetUserList();

*// Get the tickettypelist*

\_ticketTypeList = proxy.GetTicketTypeList();

}

On the other hand, the post version is executed when the user hits the "Submit" button. We will create a ticket object, map the properties to the field properties of the form, validate the objects data (see the source code of the model for detailed validation information as validation data is stored in the model classes), and persist the newly added ticket through the service to the database.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ)

[AcceptVerbs("POST")]

public ActionResult Ticket(FormCollection p\_form)

{

*// Create placeholder for the ticket to add.*

var TicketToCreate = new Ticket();

try

{

*// Map the controls of the View to the PlaceHolder.*

this.UpdateModel(TicketToCreate, new[]

{ "UserId", "TicketTypeId", "Comment", "GrantedToDate" });

*// Force Validation*

TicketToCreate.Validate();

*// First check if any validation Errors Exist on the newly*

added ticket

if (TicketToCreate.HasErrors)

{

*// View Validation Errors*

foreach(DictionaryEntry entry in

TicketToCreate.ValidationErrors)

{

ViewData.ModelState.AddModelError(entry.Key.ToString

(), entry.Value.ToString());

}

throw new InvalidOperationException();

}

else

{

*// Create the WCF Remote service and let the service*

persist the new ticket.

BtServiceRef.BtServiceClient proxy = new

BugTracking.BtServiceRef.BtServiceClient();

Ticket[] newTicketList = new Ticket[] { TicketToCreate };

proxy.PersistTickets(ref newTicketList);

}

}

catch

{

this.LoadSelectionLists();

var userList = new SelectList(\_userList, "UserId", "Username");

ViewData["UserId"] = userList;

var ticketTypeList = new SelectList

(\_ticketTypeList, "TicketTypeId", "TicketTypeDescription");

ViewData["TicketTypeId"] = ticketTypeList;

*// Show possible validation errors.*

return View(TicketToCreate);

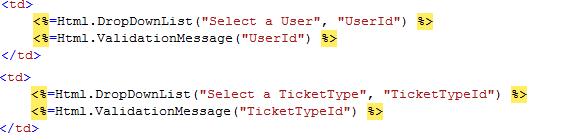
}

*// Return to main page.*

return RedirectToAction("Index");

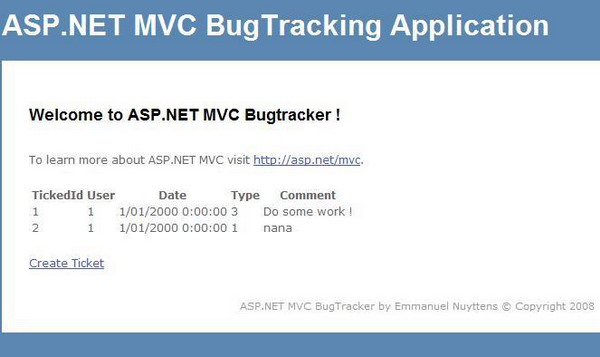
}

There are two comboboxes on the *ticket.aspx* form: one for user selection and one for ticket type selection. As shown in the code above, we load the user and ticket type records from the service and add them as ViewData of the ticket page. Next, we have to bind the ViewData as mentioned below (you can also notice how we execute the bindings for validation purpose):



***Index.aspx***

When loading the application in the browser, the index page is the first page shown. We get a list of current tickets in the database and the possibility to add a new ticket to the system:



Again, the **HomeController** is responsible for rendering the page, the action involved is:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/32237/N-Tier-development-with-ASP-NET-MVC-WCF-and-LINQ)

public ActionResult Index()

{

ViewData["Title"] = "Home Page";

ViewData["Message"] = "Welcome to ASP.NET MVC Bugtracker !";

*// Create a proxy to the WCF service.*

BtServiceRef.BtServiceClient proxy =

new BugTracking.BtServiceRef.BtServiceClient();

*// Get all tickets from the DataLayer.*

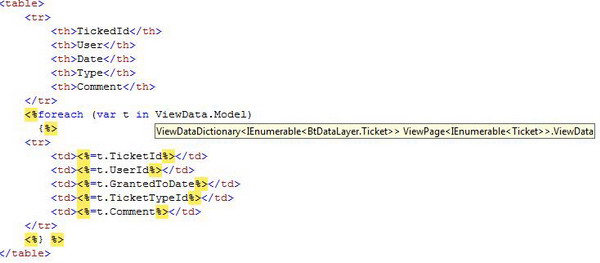
return View(proxy.GetTickets());

}

As we will show the current tickets in this page, we first create a reference to our service. This service calls theGetTickets() method of our data layer data-access class and adds it as input parameter of the view. Next, to be sure that our *Index Page* is aware of the loaded ticket-list, the BtDataLayer.Ticket type should be added as an attribute to the derived **ViewPage**:

indexpage_classdef.jpg

Finally, in the HTML code of our *index.aspx* page, we can loop through our loaded ticket list and place each ticket in a data row:



You may have noticed that I display the UserId and TickeTypeId instead of the user name and ticket type description. If you want to view, for example, the username instead of the ID, you should use: t.User.UserName, but this will return an error, as the user is not loaded with the tickets. You can change this by altering the data access method and adding data load options for the user and ticket type. In this case, the related user and ticket type objects will be loaded too.

Final Word

Voilà, that's all to mention. This article showed a possible way to separate logical layers and pass data between boundaries using new technologies such as WCF and LINQ-to-SQL. Finally, it showed a possible presentation layer, using the newest ASP.NET MVC technology. Special thanks goes to the people running the ASP.NET website[www.asp.net](http://www.asp.net/) and also special thanks goes to Beth Massi: <http://blogs.msdn.com/bethmassi/>, for explaining LINQ-to-SQL and N-Tier through WCF in a well structured and clear view.

## WCF, Entity Framework and N-Tier Solutions – Part 2

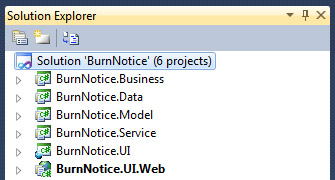
I wrote a post a few weeks ago about an issue that I ran into while programming my first WCF application. If you would like to read it that post first (it might be a good idea) you can click [here](http://www.toplinestrategies.com/dotneters/wcf/wcf-entity-framework-and-n-tier-solutions/). Since it was posted, I have noticed that a few people visit the post every day and they are probably annoyed since I identified the problem, identified the solution and then gave no explanation as to how the solution should actually be implemented. So, as promised, I am going to go over the path that we followed the solve the problem step by step. This will be a long post, but if you are trying to solve this particular problem, I hope that it is helpful.

# Create the Solution

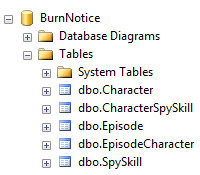
I want to create a new Burn Notice Silverlight project to keep track of all the characters and their spy skills. In case you have not read any of my previous posts and have not seen my “Burn Notice” examples (and if you don’t know what Burn Notice is) … you are missing out. Click [here](http://www.imdb.com/title/tt0810788/) to learn more. So to begin with, let’s create a new solution and then add projects to it that will house the different tiers of our application. Using Visual Studio, create an empty solution and add in the following projects (in parentheses I have added what I will call each project in my Burn Notice solution):

* Model Layer – This layer will contain the POCO objects (BurnNotice.Model)
* Data Layer – This layer will interact with the database and will contain our Entity Framework file (BurnNotice.Data)
* Business Layer – This layer calls the data layer and implements business rules if applicable (BurnNotice.Business)
* Service Layer – This layer contains the actual WCF service that can be accessed by external websites and our silverlight project (BurnNotice.Service)
* Silverlight Layer – This layer contains the silverlight project that references the service layer (BurnNotice.UI)
* Web Layer – This is a website that will host the silverlight project (BurnNotice.UI.Web)

This is what it looks like in my solution explorer:

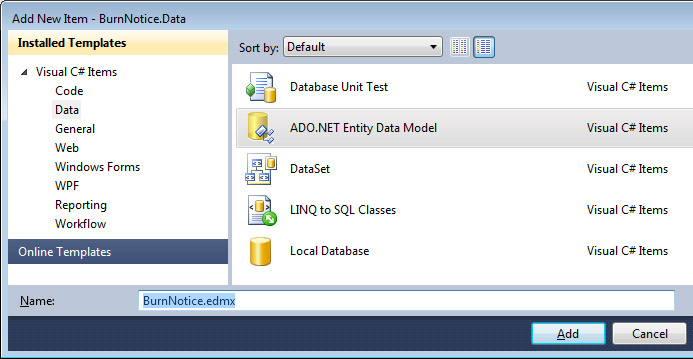


You, of course, can name your projects whatever you want as long as you know what each layer is for. Model, Data, Business etc. are just naming conventions that we have used. In my case, I have also created a database with a few tables in it. I have included a small screenshot of the table names.



# Entity Framework Model and POCO’s

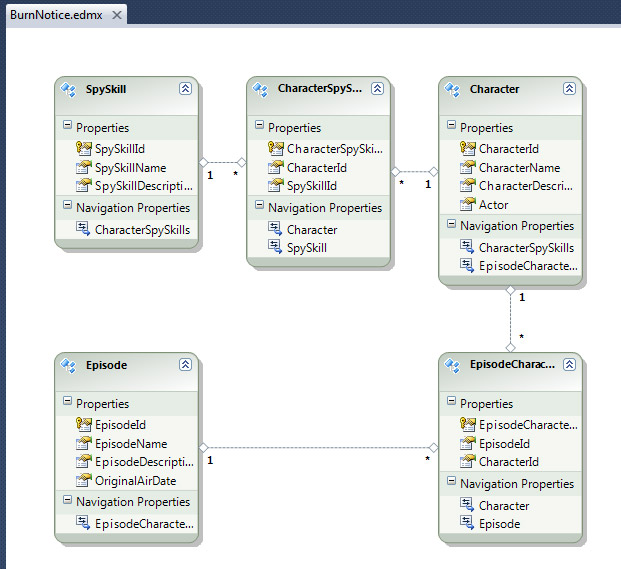
Now that your database is created, and all the table relationships set up properly, right click on the Data project in your solution and select Add > New Item … When the dialog box appears, click on “Data” in the “Installed Templates” section. Once that has been done, click on the “ADO.NET Entity Data Model”, name the file whatever you would like and then save it by clicking “Add”.



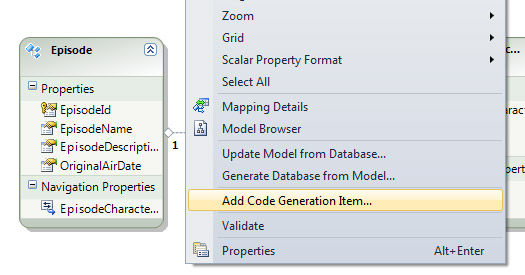
At this point, the “Entity Data Model Wizard” appears. Do the following:

1. Select “Generate From Database” on the “Choose Model Contents” page.
2. Click Next
3. Click on “New Connection” and then use the “Connection Properties” popup to build the connection string to your database. I recommend that you use “SQL Server Authentication” with an account that is set up with only the necessary permissions for your applications
4. Click on “Test Connection”, if the popup says “Test connection succeeded” then click on the “OK” button
5. Click on the radio button labeled “Yes, include the sensitive data in the connection string” (this can be encrypted later if you wish)
6. Click the checkbox labeled “Save entity connection settings in App.Config as” and then give the connection a name (whatever you want to call it, but remember what it is)
7. Click Next
8. Select all the database objects that you want to include in the Entity Framework (Most likely tables)
9. Enter a name for the Model Namespace (whatever you want to call it)
10. Click Finish

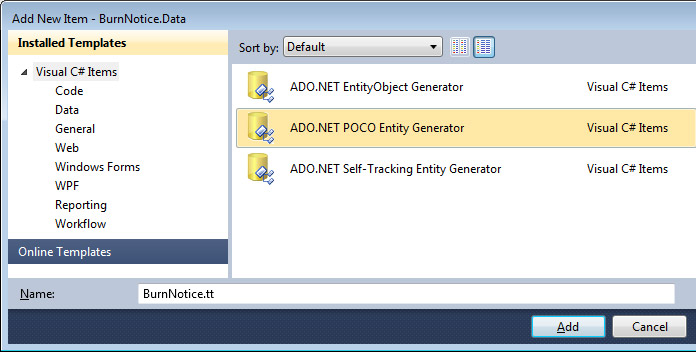
Now that the Entity Framework Model is created, we are ready to get to the meat of the presentation which is creating POCOs (Plain Old CLR Objects) which can then be exposed over a WCF service without exposing the Entity Framework itself and all that goes with it. Once you are done, your Entity Framework Model should look something like this:



So now that you are looking at the Entity Framework file that you just created, right click on the canvas of the Entity Framework. You will see several options in the menu that pops up. We want to select “Add Code Generation Item …”.

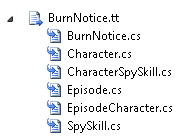


This will bring up another window showing the “Installed Templates” as shown in the screenshot below. Once the “Add New Item” window pops us, we want to select the template called “ADO.NET POCO Entity Generator”. This also can be called whatever you like although I tend to keep the name the same as what I chose for the Entity Framework itself.



Incidentally, if you cannot find this template installed in your copy of Visual Studio, you can find instructions on searching for and installing templates [here](http://weblogs.asp.net/scottgu/archive/2010/05/03/visual-studio-2010-extension-manager-and-the-new-vs-2010-powercommands-extension.aspx). Once you save the “ADO.NET POCO Entity Generator”, two new files will be created in your data project. In my project, they are: “BurnNotice.Context.tt” and “BurnNotice.tt”.

At this point, if you try to build the project it should compile without any problems, but there are a couple more things to notice. First of all, underneath the new template files (those with a .tt extension), you will see that several new classes have been added to your data project, as shown in the graphic below.

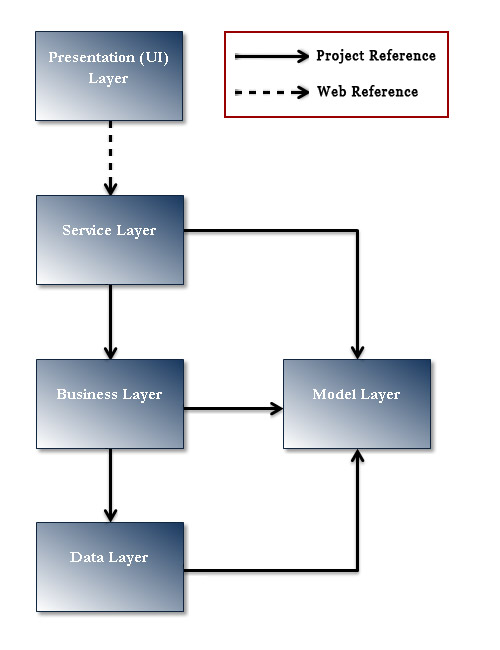


Each of these new classes maps to an object in your Entity Framework model, which, in turn, map to a table in your database. These files should not be edited directly (they are autogenerated each time that the template file is saved), but if you open one up you will see that it is a fairly standard class with properties defining each column in the table. You will also notice that they are partial classes. That is not relevant to this post, but I have found it useful in other situations.

# N-Tier Structure

Now, everything is working, however, any data access would have to go through directly to our data layer. Problems start to crop up when you try to start making it into a fully functional n-Tier solution. Mostly this is caused when you start adding in the references to the various projects in the solution. Typically making calls to the database in an n-Tier solution follows the following pattern.

UI Layer calls the Business Layer which then calls the Data Layer. Once the data has been retrieved from the database, the Data Layer passes it back to the Business Layer, where any rules are applied. The Business Layer then sends the data back to the UI Layer where it is presented to the user. In our project we also have the service layer sitting between the UI Layer and the Business Layer, but the concept is the same. However, if you want to pass actual objects or collections of objects back and forth between these layers (as opposed to say … an array or a data table), all of your layers need to have access to the place in which the objects are defined. If they are defined in the Data Layer (as our test project is currently set up) then all other layers need to have a reference to it. You will find that this makes it very easy to have circular reference errors in your project and it also makes it harder to use those object with a service. At TopLine, we typically have the Model project sitting off to the side and referenced by all other projects in the solution as demonstrated in the screenshot below:



By doing this, all of the projects can pass around the commonly defined objects found in the model project. It is here where the rubber hits the road, so to speak. For an N-Tier structure to work with WCF and Entity Framework, we have created the POCO’s and we will now move them over to our model project so that they can be the common objects that we are passing back and forth. We then have the benefits of using the Entity Framework for data access, but using the POCO’s to pass around and expose externally via the service.

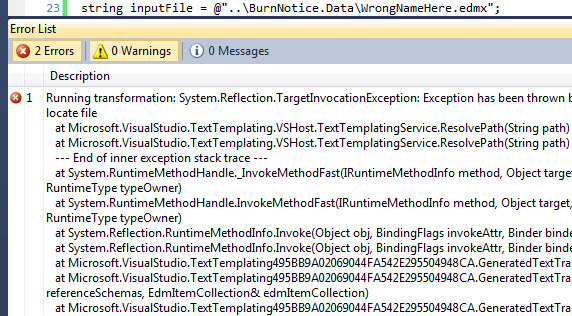
To move our POCO’s from the Data Project to the Model Project is fairly easy to do. Right click on the template in the Data Project that does NOT have the word “context” in the name (BurnNotice.tt in my example). Select “Cut” from the context menu, then right click on the Model project and select paste. Both the template and all of its associated classes should now have moved to the Model project. Our only problem now is that by moving the template we have broken the link between it and the Entity Framework model upon which it depends to create the POCO classes. Luckily, this is easy to fix. To do so, open the template file that is now in the Model project. Incidentally, there is currently no intellisense support for template files in Visual Studio. It will just be plain old black and white. Don’t let that fool you though. These templates are powerful tools. Once it is open … look for the line of code that reads something like this:

|  |  |
| --- | --- |
| 1 | string inputFile = @"BurnNotice.edmx"; |

And change it to a path where it can find the Entity Framework model over in the data project, like so:

|  |  |
| --- | --- |
| 1 | string inputFile = @"..\BurnNotice.Data\BurnNotice.edmx"; |

When you click save, you will know immediately if this worked. If it did, the class files for your database objects will be regenerated as they should be. If the path that you entered was incorrect, however, you will receive a huge ugly error that looks like this:



OK … a few more things to do and we are home free. First of all, there is another item that we need to move to the Model project and that is the interface that defines our BurnNoticeService. This needs to be in our Model project for the same reason that the POCO’s do. The interface is going to be implemented by classes in the Business and Service layers of the solution. In order for that to happen, all projects need to have access to it. Just as we moved the BurnNotice template into the Model project, right click on the Interface in the Service Project (“IBurnNoticeService” in my project), select “Cut”, then right click on the Model Project and select “Paste”. Once it has been moved, you will need to make a couple more changes to it so that the solution will compile. First of all, open the interface and change the namespace of it from the Service Project namespace to the Model namespace. Second, you will need to add a reference to the “System.ServiceModel” namespace in your Model Project (This was automatically added to the service project when you created it, but it has to be manually added to the Model Project).

# Project References

Now we are ready to start adding in our project references. Just going down the list, your references should look like this:

**Data Project – References To**  
Model Project

**Business Project – References To**  
Data Project  
Model Project

**Service Project – References To**  
Business Project  
Model Project

**Model Project – References To**  
No Project References

At this point, we need to make one more change to a template file. This is necessary because the Context template file should be pointing at our POCO objects, which are now in a different project. We already have the reference to the Model Project added to our Data Project, so now we just need a reference to the namespace in the Context class itself. Go back to the Data project where you have the Entity Framework Model and a template file that includes the word “Context” in the title. Open it and find the following code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | //--------------------------------------------------------------------  //  //     This code was generated from a template.  //  //     Changes to this file may cause incorrect behavior and will be lost if  //     the code is regenerated.  //  //--------------------------------------------------------------------    using System;  using System.Data.Objects;  using System.Data.EntityClient; |

Now add a Using statement here that points to the Model Project, like So:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | //--------------------------------------------------------------------  //  //     This code was generated from a template.  //  //     Changes to this file may cause incorrect behavior and will be lost if  //     the code is regenerated.  //  //--------------------------------------------------------------------    using System;  using System.Data.Objects;  using System.Data.EntityClient;  using BurnNotice.Model; |

In the same template find the method named “WriteLazyLoadingEnabled. It looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | private void WriteLazyLoadingEnabled(EntityContainer container)  {     string lazyLoadingAttributeValue = null;     string lazyLoadingAttributeName = MetadataConstants.EDM\_ANNOTATION\_09\_02 + ":LazyLoadingEnabled";     if(MetadataTools.TryGetStringMetadataPropertySetting(container, lazyLoadingAttributeName, out lazyLoadingAttributeValue))     {         bool isLazyLoading = false;         if(bool.TryParse(lazyLoadingAttributeValue, out isLazyLoading))         {  #>          this.ContextOptions.LazyLoadingEnabled = <#=isLazyLoading.ToString().ToLowerInvariant()#>;  <#+         }     }  }  #> |

and add one line to it (highlighted below) This will prevent your project from creating duplicate proxy classes that interfere with execution.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | private void WriteLazyLoadingEnabled(EntityContainer container)  {     string lazyLoadingAttributeValue = null;     string lazyLoadingAttributeName = MetadataConstants.EDM\_ANNOTATION\_09\_02 + ":LazyLoadingEnabled";     if(MetadataTools.TryGetStringMetadataPropertySetting(container, lazyLoadingAttributeName, out lazyLoadingAttributeValue))     {         bool isLazyLoading = false;         if(bool.TryParse(lazyLoadingAttributeValue, out isLazyLoading))         {  #>          this.ContextOptions.LazyLoadingEnabled = <#=isLazyLoading.ToString().ToLowerInvariant()#>;          this.ContextOptions.ProxyCreationEnabled = false;  <#+         }     }  }  #> |

Once you save it, the Context class will be recreated with the Model Project reference included in it.

# WCF Service

We are now ready to wire up the WCF service and to have it function in an N-Tier manner. Obviously this can be a lot more complex than I will demonstrate here, but for the sake of simplicity, I will add a new class to the Data, Business and Service Projects. In my case I will simply call each one “BurnNotice.cs”, you may call it whatever you like. Next, I will open up the BurnNotice interface and put in some methods that I want used every time this interface is implemented. Even though this interface is in the Model project, we need to add the Service Contract attributes to it just as we would do if it were still in the service project, mine looks like this:

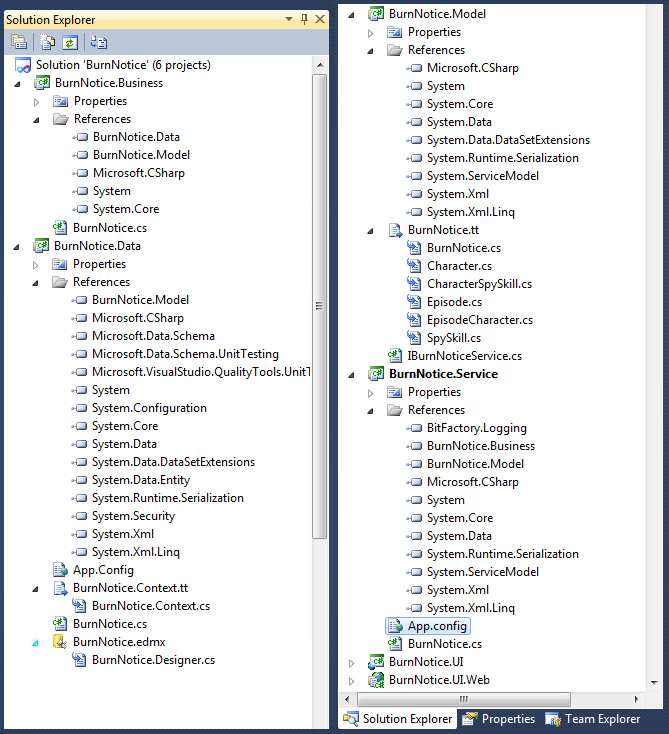
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Runtime.Serialization;  using System.ServiceModel;  using System.Text;    namespace BurnNotice.Model  {      [ServiceContract]      public interface IBurnNoticeService      {          [OperationContract]          Character Character\_GetById(int characterId);            [OperationContract]          IEnumerable<Character> Characters\_GetAll();            [OperationContract]          IEnumerable<Character> Characters\_GetByEpisode(int episodeId);            [OperationContract]          IEnumerable<Character> Characters\_GetBySpySkill(int spySkillId);            [OperationContract]          IEnumerable<SpySkill> SpySkills\_GetByCharacterId(int characterId);            [OperationContract]          Episode Episode\_GetById(int episodeId);            [OperationContract]          IEnumerable<Episode> Episode\_GetAll();      }  } |

Now that we have the interface ready, I will go into each instance of the “BurnNotice.cs” class that I created in the Data, Business and Service projects and implement the interface. In case you are unfamiliar with interface implementation, in C# it is done by putting a colon and then the name of the interface that you want to implement after the class name in the class file (For a technical, but very good explanation of interfaces, how they work and why they are desirable click [here](http://www.c-sharpcorner.com/UploadFile/rmcochran/csharp_interrfaces03052006095933AM/csharp_interrfaces.aspx). You have to have an interface for WCF to work properly anyway, but using it in the rest of the application is an excellent idea too.) It is implemented like this:

|  |  |
| --- | --- |
| 1 | public class BurnNotice : IBurnNoticeService |

If you were to try to compile the solution at this point, however, it will fail because even though you have added a reference to the interface, it has not technically been implemented. A quick shortcut to remedy this is to right click on the name of the interface after the colon in the class and then select “Implement Interface” > “Implement Interface”. This will add all of the required methods to the class which is implementing the interface, which you can then alter to meet your project’s needs. Do this on each of the classes that you just created. By the way, technically speaking you do not need the interface in the Business and Data projects to create methods that will return data from the database, but using interfaces is very helpful in making sure that all required functionality is present in your application. The Service Project on the other hand is a different matter. In order to expose it as a WCF service you must have the interface defined, which is what will allow any application that consumes your service to be aware of the methods and the formats of the data that it will be dealing with. I prefer then to implement the interface in all classes that will use the data.

I mentioned in my last post on this subject that I found many great tutorials, but none showed exactly what I needed to get an n-tier solution up and running. In fairness to them, I don’t think that was really their focus. Just to make sure that you can see the entire project structure, in the screenshot below you will see an almost complete graphic of my Solution Explorer. My hope is that this will prevent anyone from being frustrated at not knowing which project references go where and/or which namespaces need to be included with which project.



# Implementation

After all of this, we can now use the service, but have it call our different layers as we would in a standard n-tier solution. In the App.Config file, you will need to add in the connection string to your database in an Entity Framework approved format like this:

|  |  |
| --- | --- |
| 1  2  3 | <connectionStrings>      <add name="BurnNotice\_Database" connectionString="metadata=[res://](NULL)\*/BurnNotice.csdl|[res://](NULL)\*/BurnNotice.ssdl|[res://](NULL)\*/BurnNotice.msl;provider=System.Data.SqlClient;provider connection string=&quot;Data Source=LocalHost;Initial Catalog=BurnNotice;Persist Security Info=True;User ID=fiona;Password=Sam!Axe\*;MultipleActiveResultSets=True&quot;" providerName="System.Data.EntityClient" />  </connectionStrings> |

From the Service Layer then, the code to call the Business Layer looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Runtime.Serialization;  using System.ServiceModel;  using System.Text;  using BurnNotice.Model;  using BurnNotice.Business;  using BitFactory.Logging;    namespace BurnNotice.Service  {      public class BurnNotice : IBurnNoticeService      {            #region IBurnNoticeService Members            public Character Character\_GetById(int characterId)          {              Business.BurnNotice bn = new Business.BurnNotice();              return bn.Character\_GetById(characterId);          }            public IEnumerable<Character> Characters\_GetAll()          {              Business.BurnNotice bn = new Business.BurnNotice();              return bn.Characters\_GetAll();          }            public IEnumerable<Character> Characters\_GetByEpisode(int episodeId)          {              Business.BurnNotice bn = new Business.BurnNotice();              return bn.Characters\_GetByEpisode(episodeId);          }            public IEnumerable<Character> Characters\_GetBySpySkill(int spySkillId)          {              Business.BurnNotice bn = new Business.BurnNotice();              return bn.Characters\_GetBySpySkill(spySkillId);          }            public IEnumerable<SpySkill> SpySkills\_GetByCharacterId(int characterId)          {              Business.BurnNotice bn = new Business.BurnNotice();              return bn.SpySkills\_GetByCharacterId(characterId);          }            public Episode Episode\_GetById(int episodeId)          {              Business.BurnNotice bn = new Business.BurnNotice();              return bn.Episode\_GetById(episodeId);          }            public IEnumerable<Episode> Episode\_GetAll()          {              Business.BurnNotice bn = new Business.BurnNotice();              return bn.Episode\_GetAll();          }            #endregion      }  } |

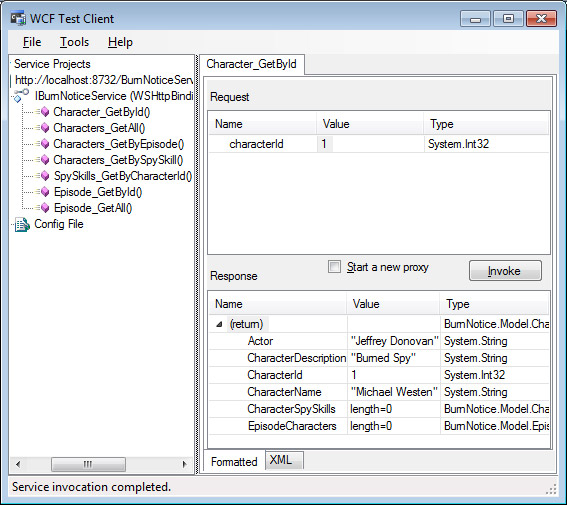
From the Business Layer, where we have a reference to the Data Layer, we can call the analogous methods in the Data Layer that will actually run queries against the Entity Framework. Once the data comes back and before we send it along to the service layer, we can implement any rules that we want enforced (but don’t necessarily want the data changed in the database) and make changes to the data accordingly. In my example below, you will see that I have put in code to append the text “: Bad Guy” if the description of the character uses the word “Evil” when I am returning a character’s data from the database. This is dumb I know, but it is only for illustration. I’m assuming that real life example would make more sense:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using BurnNotice.Model;  using BurnNotice.Data;    namespace BurnNotice.Business  {      public class BurnNotice : IBurnNoticeService      {          #region IBurnNoticeService Members            public Character Character\_GetById(int characterId)          {              Data.BurnNotice bn = new Data.BurnNotice();              Character ch = bn.Character\_GetById(characterId);                if (ch.CharacterDescription.ToLower().Contains("evil"))              { ch.CharacterDescription += ": Bad Guy"; }                return ch;          }            public IEnumerable<Character> Characters\_GetAll()          {              Data.BurnNotice bn = new Data.BurnNotice();              return bn.Characters\_GetAll();          }            public IEnumerable<Character> Characters\_GetByEpisode(int episodeId)          {              Data.BurnNotice bn = new Data.BurnNotice();              return bn.Characters\_GetByEpisode(episodeId);          }            public IEnumerable<Character> Characters\_GetBySpySkill(int spySkillId)          {              Data.BurnNotice bn = new Data.BurnNotice();              return bn.Characters\_GetBySpySkill(spySkillId);          }            public IEnumerable<SpySkill> SpySkills\_GetByCharacterId(int characterId)          {              Data.BurnNotice bn = new Data.BurnNotice();              return bn.SpySkills\_GetByCharacterId(characterId);          }            public Episode Episode\_GetById(int episodeId)          {              Data.BurnNotice bn = new Data.BurnNotice();              return bn.Episode\_GetById(episodeId);          }            public IEnumerable<Episode> Episode\_GetAll()          {              Data.BurnNotice bn = new Data.BurnNotice();              return bn.Episode\_GetAll();          }            #endregion      }  } |

So, it is in the data layer then that the magic happens. In the Data Layer, we can use calls to the Entity Framework Model using its data context (called BurnNotice\_Database() in the code below) and utilize all of the benefits of the Entity Framework, but as far as the objects that we are actually passing around, they are the POCO’s and not the actual Entity Framework objects. So, once the call reaches the BurnNotice class in the Data Layer, the calls to the database are done like so:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using BurnNotice.Model;    namespace BurnNotice.Data  {      public class BurnNotice : IBurnNoticeService      {          #region IBurnNoticeService Members            public Character Character\_GetById(int characterId)          {              using (var context = new BurnNotice\_Database())              {                  Character character = context.Characters.Where(c => c.CharacterId == characterId).FirstOrDefault();                  return character;              }          }            public IEnumerable<Character> Characters\_GetAll()          {              using (var context = new BurnNotice\_Database())              {                  return context.Characters.ToList();              }          }            public IEnumerable<Character> Characters\_GetByEpisode(int episodeId)          {              using (var context = new BurnNotice\_Database())              {                  var characters = from c in context.Characters                                   join e in context.EpisodeCharacters on c.CharacterId equals e.CharacterId                                   where e.EpisodeId == episodeId                                   select c;                      return characters;              }          }            public IEnumerable<Character> Characters\_GetBySpySkill(int spySkillId)          {              using (var context = new BurnNotice\_Database())              {                  var characters = from c in context.Characters                                   join s in context.CharacterSpySkills on c.CharacterId equals s.CharacterId                                   where s.SpySkillId == spySkillId                                   select c;                      return characters;              }          }            public IEnumerable<SpySkill> SpySkills\_GetByCharacterId(int characterId)          {              using (var context = new BurnNotice\_Database())              {                  var spySkills = from s in context.SpySkills                                  join c in context.CharacterSpySkills on s.SpySkillId equals c.SpySkillId                                  where c.CharacterId == characterId                                  select s;                      return spySkills;              }          }            public Episode Episode\_GetById(int episodeId)          {              using (var context = new BurnNotice\_Database())              {                  Episode episode = context.Episodes.Where(e => e.EpisodeId == episodeId).FirstOrDefault();                  return episode;              }          }            public IEnumerable<Episode> Episode\_GetAll()          {              using (var context = new BurnNotice\_Database())              {                  return context.Episodes.ToList();              }          }            #endregion      }  } |

If you set the Service Project to be the Startup Project and then run the solution, the WCF test client will come up and, as shown in the screenshot below, I am able to see a list of all my service methods and then invoke them and receive data back from the service.



In [part three](http://www.toplinestrategies.com/dotneters/net/wcf-entity-framework-and-n-tier-solutions-part-3/) of this tutorial, I show how to do that last part of this process, which is to call the service from another project (in this case, a Silverlight Application). I hope that will be useful as well, but Part 2 should be enough to show how using POCO’s enables us to implement a full n-tier solution with WCF and the Entity Framework.

# \*\*\*\*WCF databinding using ADO.NET

<http://www.codeproject.com/Articles/45903/WCF-databinding-using-ADO-NET>

* [**Download source code - 22.7 KB**](http://www.codeproject.com/KB/WCF/WCFDatabinding/WCFDatabinding.zip)

## Introduction

When I wanted to learn creating WCF Web Services, I thought it would not be a bad idea to do databinding using ADO.NET. Many articles I went through used LINQ to SQL. Although this was helpful, I decided to do a WCF app that retrieves data off of a database using plain ADO.NET. When I looked, there weren't any articles that would demonstrate that. This is a very basic example like a Hello World for retrieving data from a SQL Server database.

## Background

As we all know, the DataContract attribute class is used to mark types you write as participating in the WCF serialization via the DataContractSerializer. Marking your classes with this attribute ensures that they can be sent to and from disparate clients in an efficient manner. This is automatically done when we use LINQ classes (.dbml) when you mark the property of the class for "uni-directional serialization".

We will try to retrieve all the records from a table in a database. I used my company database in this example. But you can change it to be the Northwind database by just changing the query.

The project involves creating a "WCF Service application" and a web client to test the service.

## Using the code

We define the interface with two operation contracts as below:

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/45903/WCF-databinding-using-ADO-NET)

*//IMyService.cs*

[OperationContract]

Shipper GetShipper(int shipperID);

[OperationContract]

Shipper saveShipper(Shipper shipper);

Now let's define the Data Contract with the two data members which need to be serialized.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/45903/WCF-databinding-using-ADO-NET)

*//Define the data contract IMyService.cs*

[DataContract]

public class Shipper

{

[DataMember]

public int ShipperID

{

get; set;

}

[DataMember]

public DataSet dsTable

{

get;set;

}

### Web client

Create a regular ASP.NET website and add a web reference to the Web Service just created, and invoke the methods.

Bottom of Form