Building a Data Access Layer – The Demo Application

By Damon Armstrong

This PDF walks through the code required to build a DAL in C# and accompanies the Simple-Talk article *Designing and Building a Data Access Layer* by Damon Armstrong and the *DALDemoApp* zip file containing the application.

The application is fairly simple, a two page web app that allows you to view / delete a list of people on one page and to add / edit those people on another. Before jumping into the code, I want to make sure everyone has a good overview of all the assemblies and classes in the solution and how they fit together. Figure 7 contains a solution diagram to help convey the structure of the application.

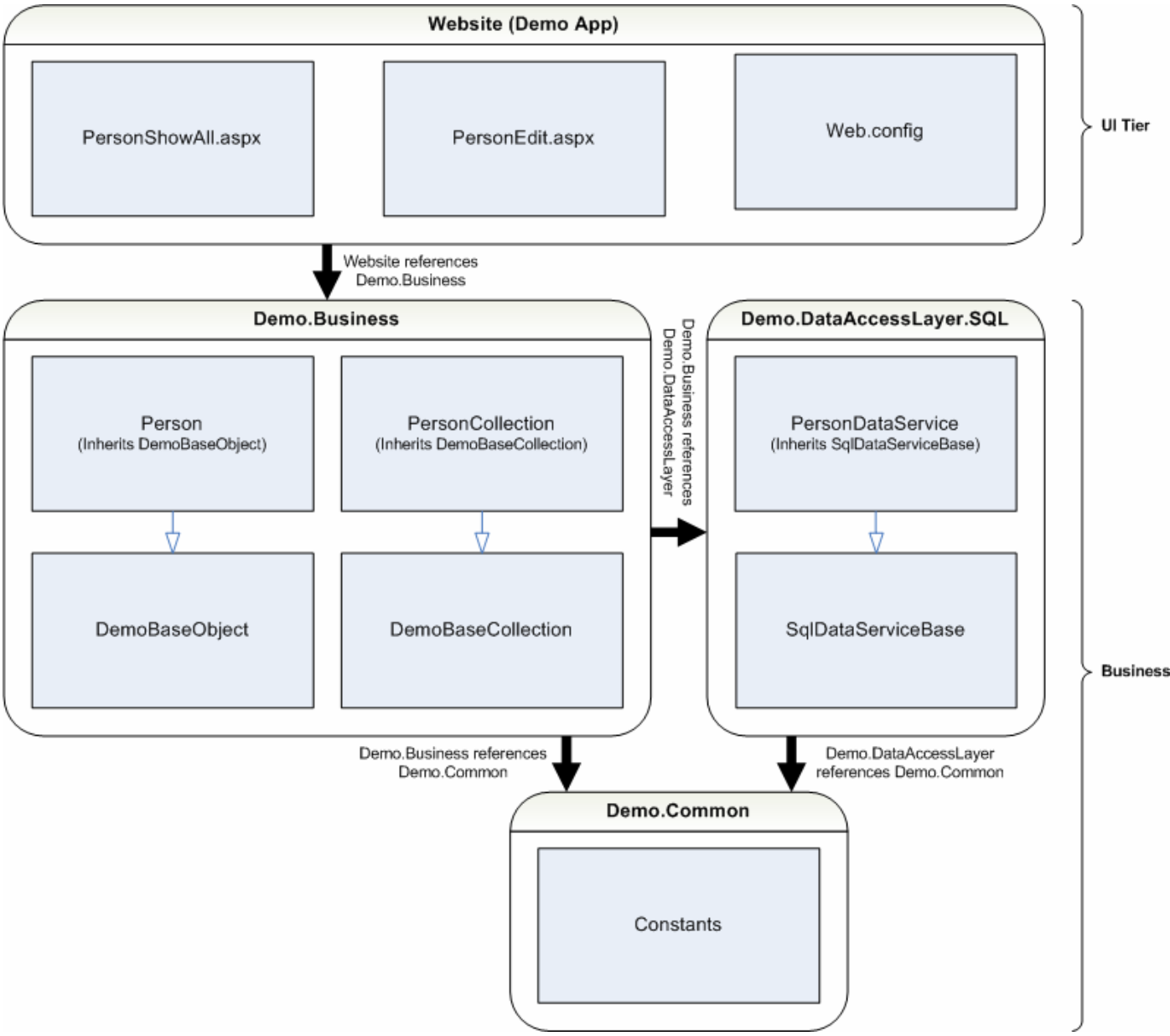

Figure 7 – Solution Diagram outlining the various assemblies and classes in the application

Table 1 further describes each item and its purpose:

Table 1 – Description of items in the Solution Diagram (Figure 7)

Name	Description
Demo.Common	Assembly containing shared constants for the Demo.Business and Demo.DataAccessLayer assemblies
Constants	Contains a list of constants to define "null” values for

	native .NET types. Since .NET value-types cannot be set to null, these values help the application mark properties as being null by assigning them a special "null value"
Demo.DataAccessLayer (SQL)	Data Access Layer Assembly for SQL Server.
DataServiceBase	Contains helper methods for SQL Server to establish a connection, manage a transaction, and execute stored procedures. This class acts as the base class for all Data Service Classes in the DAL.
PersonDataService	Data Service Class containing data access methods specific to the Person business object
Demo.Business	Business Objects Assembly
DemoBaseObject	Contains helper methods for mapping data from the DAL into the business object and pulling data from a DataSet. This class acts as the base class for all business objects.
DemoBaseCollection	Contains helper methods for mapping data from the DAL into a collection of business objects. This class acts as the base class for all business collections.
Person	Contains methods and properties specific to the Person business object.
PersonCollection	Represents a collection of Person objects.
Website (Demo App)	Web Application
Web.config	Stores the connection string to the Database
PersonShowAll.aspx	Displays a list of all people in the application. You may click on the Edit link next to a person's name to edit the person or on the Add link at the bottom of the page to add a new person. You may also delete users by selecting checkboxes next to their names and clicking on the delete link.
PersonEdit.aspx	Displays entry field allowing you to add or edit a user.

Now let's get into the code, starting at the bottom of the diagram and working our way up.

Constants class - Demo.Constants Assembly

Databases allow for the concept of null -- the idea that value does not exist. This is a bit of an issue for your code because native .NET types (e.g. int, double, DateTime, etc) have no concept of null. If you have a value type in your application, then it's got a value, that's all there is to it. So you have to simulate the concept by defining a value for the type that represents null. I call them null values for lack of a better term. In your application, you have to setup your code to determine whether a value type is null based on its null value and to process the type accordingly.

Your main objective in choosing a null value for a type is to choose a value that makes sense and is highly unlikely to be used. For example, let's decide on a value for the int value type. At first, you may say 0 would be a good choice for the null value since zero is indicative of nothing. The problem is that zero actually doesn't mean nothing, it means zero, and it comes up a lot. There is a big difference between having 0 dollars in your bank account and have a null value. Null means that the actual value is not known, so you could be a millionaire. Zero means you're broke. See the difference? So you're better off choosing something else. I recommend using -2147483648, also known as the minimum value for an int type, for two reasons. First, the chances of your application actually needing to use the number -2147483648 are infinitesimally small. Second, there is a nice constant that defines the value called int.MinValue so you don't have to remember it.

The Constants class exposes a set of public static fields that contain null values for the various .NET types commonly used in applications. As static fields, they are not truly constants, but we use them like constants in the application. Why aren't they actual constants? Some .NET types, like DateTime and Guid, cannot be a constant, so I chose to expose everything as a public field for brevity. You can take this a step further and make them read-only properties if you want. Listing 2 contains the code for the Constants class in the Demo.Common assembly:

Listing 2 – Constants class

```
using System;

namespace Demo.Common
{
    public sealed class Constants
    {
        //Null values
        public static DateTime NullDateTime = DateTime.MinValue;
        public static decimal NullDecimal = decimal.MinValue;
        public static double NullDouble = double.MinValue;
        public static int NullInt = int.MinValue;
        public static long NullLong = long.MinValue;
        public static float NullFloat = float.MinValue;

        //Types without MinValue properties
        public static string NullString = string.Empty;
        public static Guid NullGuid = Guid.Empty;
```

```
        //Helpful values
        public static DateTime SqlMaxDate = new DateTime(9999, 1, 3, 23,
59, 59);
        public static DateTime SqlMinDate = new DateTime(1753, 1, 1, 00,
00, 00);
    }
}
```

As you can see, the field names (bolded in the listing) follow a naming convention of Null<TypeName>, and most use the MinValue of their types as their null value. But the string and Guid types do not have a MinValue property, so we have to choose the values for them a bit differently. In my experience, string.Empty works well for strings since an empty string and a null string normally equate to the same thing. If, for some reason, you need to differentiate between the two the you can use a random assortment of garbled characters. For the Guid's null value we use Guid.Empty because it equates to 00000000-0000-0000-0000-000000000000, which is easily identifiable to the naked eye. In theory, you could use any Guid as the null value and the chances of it actually coming up in your application are about one in a gazillion, but when you are debugging it's much easier to identify a big long string of zeros than a completely random Guid.

There are also two constants at the bottom of the listing that are not related to null values, but are helpful nonetheless. SQL Server can only handle dates between 1/1/1753 12:00:00 AM and 1/3/9999 11:59:59 PM. I can't ever remember these dates, so I put them in the Constants class in case I ever needed them. And now you have them too in case you want to setup validation on date inputs to ensure SQL server can handle them. Next, we'll take a look at the much more code intensive DataServiceBase class.

DataServiceBase class – Demo.DataAccessLayer Assembly

The DataServiceBase class is a base class for Data Service classes in the DAL. It is responsible for handling connecting to the database, maintaining a transaction over multiple calls, setting up command parameters, executing commands, and populating DataSets with query results. It's a fairly lengthy piece of code, so I'll begin the code listing and comment intermittently where appropriate.

Listing 3 – DataServiceBase class

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
using Demo.Common;

namespace Demo.DataAccessLayer
{
    public class DataServiceBase
    {
```

```
////////////////////////////////////////////////////////////////////////
        // Fields

////////////////////////////////////////////////////////////////////////
        private bool _isOwner = false;   //True if service owns the
transaction
        private SqlTransaction _txn;     //Reference to the current
transaction



////////////////////////////////////////////////////////////////////////
        // Properties

////////////////////////////////////////////////////////////////////////
        public IDbTransaction Txn
        {
            get { return (IDbTransaction)_txn; }
            set { _txn = (SqlTransaction)value; }
        }



////////////////////////////////////////////////////////////////////////
        // Constructors

////////////////////////////////////////////////////////////////////////

        public DataServiceBase() : this(null) { }


        public DataServiceBase(IDbTransaction txn)
        {
            if (txn == null)
            {
                _isOwner = true;
            }
            else
            {
                _isOwner = false;
                _txn = (SqlTransaction)txn;
            }
        }
```

There are two ways you can use a Data Service Class in your business object code – without a transaction, or with a transaction – and there are constructors for each option. The first is a parameter-less constructor that creates a new DataServiceBase object without a transaction. Notice that it simply calls the second constructor and passes in null for the transaction. The second constructor allows you to pass in a transaction, and it does a quick check to see if the transaction is null. If the transaction passed into the constructor is null, it sets the _isOwner field to false. The _isOwner field is a Boolean value that lets the class know that it is not running as part of a transaction and is responsible for opening and closing its own database connection (i.e. it owns the connection so it has to manage

it). If the transaction passed to the constructor is not null, then it sets _isOwner to false and stores a reference to the transaction in the _txn field. This lets the class know to use the connection from the transaction and not to close it after executing a command or query because some other piece of code "owns" the connection.

If you need to access the current transaction from outside the class, the DataServiceBase class exposes the current transaction via the Txn property. Internally, the _txn field stores the transaction as a SqlTransaction, but notice the Txn property exposes the SqlTransaction as an IDbTransaction. This goes back to the concept of not using database specific code in the DAL.

Listing 3 cont. – DataServiceBase class

```
//////////////////////////////////////////////////////////////////////
        // Connection and Transaction Methods
//////////////////////////////////////////////////////////////////////

        protected static string GetConnectionString()
        {
            return
ConfigurationManager.ConnectionStrings["DB"].ConnectionString;
        }


        public static IDbTransaction BeginTransaction()
        {
            SqlConnection txnConnection =
                new SqlConnection(GetConnectionString());
            txnConnection.Open();
            return txnConnection.BeginTransaction();
        }
```

The GetConnectionString method, as you may have guessed, returns the connection string used to connect to the SQL database. It does so by acquiring the connection string for the DB item in the <connectionStrings> section of the Web.config. The method is static because it does not require an object instance to function and we need to use it in the BeginTransaction method, which is also static. The BeginTransaction method is a helper function that allows you to easily create a new transaction to pass into a Data Service class. It begins by creating a new new SqlConnection using the GetConnectionString method described above, and returns the SqlTransaction object from the BeginTransaction method off the connection.

Listing 3 cont. – DataServiceBase class

```
//////////////////////////////////////////////////////////////////////
        // ExecuteDataSet Methods
//////////////////////////////////////////////////////////////////////

        protected DataSet ExecuteDataSet (string procName,
```

```
            params IDataParameter[] procParams)
        {
            SqlCommand cmd;
            return ExecuteDataSet(out cmd, procName, procParams);
        }


        protected DataSet ExecuteDataSet(out SqlCommand cmd, string
procName,
            params IDataParameter[] procParams)
        {
            SqlConnection cnx = null;
            DataSet ds = new DataSet();
            SqlDataAdapter da = new SqlDataAdapter();
            cmd = null;

            try
            {
                //Setup command object
                cmd = new SqlCommand(procName);
                cmd.CommandType = CommandType.StoredProcedure;
                if (procParams != null)
                {
                    for (int index = 0; index < procParams.Length;
index++)
                    {
                        cmd.Parameters.Add(procParams[index]);
                    }
                }
                da.SelectCommand = (SqlCommand)cmd;

                //Determine the transaction owner and process
accordingly
                if (_isOwner)
                {
                    cnx = new SqlConnection(GetConnectionString());
                    cmd.Connection = cnx;
                    cnx.Open();
                }
                else
                {
                    cmd.Connection = _txn.Connection;
                    cmd.Transaction = _txn;
                }

                //Fill the dataset
                da.Fill(ds);
            }
            catch
            {
                throw;
            }
            finally
            {
                if(da != null) da.Dispose();
                if(cmd != null) cmd.Dispose();
                if (_isOwner)
```

```
            {
                cnx.Dispose(); //Implicitly calls cnx.Close()
            }
        }
        return ds;
    }
```

Whenever you need to execute a query that returns relational data, you use the ExecuteDataSet methods. In this implementation there are two different version of the method, both of which deal with stored procedures. If you want the ability to execute ad-hoc SQL, then you can add your own ExecuteDataSet method that accepts an ad-hoc SQL string.

The first version of the ExecuteDataSet method accepts two parameters: procName is a string that contains the stored procedure name, and procParams is a parameter array of IDataParameter objects that allows you to pass in any number of command parameters to the stored procedure. It returns a DataSet containing query results from the stored procedure. You should use the first version of the method in your business object code when you need to acquire return values and output parameters from the command object after it executes. For the most part, you will not need to return the command object when executing queries because the query results themselves are normally sufficient. After defining the cmd variable, the first version of the ExecuteDataSet simply calls down to the second version and passes the cmd variable in as an out parameter. The out modifier is similar to the ref modifier in that changes to the parameter inside the method are retained when execution returns to the caller, but it implies that the value passed into the method is not used.

The second version of the ExecuteDataSet method has one additional parameter, an out SqlCommand parameter named cmd. As mentioned before, the cmd parameter allows you return the SqlCommand object executed inside the method so you can query it for output parameters and return values. The method begins by declaring three variables: cnx is a SqlConnection used to connect to the database, ds is the DataSet that stores the query results and is ultimately returned as the result of the ExecuteDataSet method, and da is a SqlDataAdapter that fills ds with query results from the stored procedure. It then sets cmd to null to avoid getting a use of unassigned out parameter 'cmd' compiler error.

After the variable declarations, the ExecuteDataSet method jumps into a Try, Catch, Finally block. The finally portion of the block helps gracefully handle the disposal of database objects in the event an error occurs, and the Catch section simply re-throws the error so you can handle it accordingly in your application. In the Try section, the method begins by creating a new SqlCommand object and setting its CommandType to CommandType.StoredProcedure. This lets the command object know it's executing a stored procedure and not ad-hoc SQL. If there are any parameters for the stored procedure in procParams, the method iterates through the procParams array and adds the parameters to the command object. It then assigns the command to be the SelectCommand on the DataAdapter.

Next, the method checks the _isOwner property to determine if it owns the connection or should use an existing transaction. If the data service owns the connection, then it creates a new SqlConnection using the GetConnectionString method, assigns the connection to the cmd.Connection property, then opens the connection so it is ready for the command to be executed. If the data service does not own the connection, then it uses the connection information from the active transaction reference stored in the _txn field to setup the command's connection. It also assigns cmd.Transaction to the transaction stored in _txn so the command knows to run as part of the transaction.

Once the command has been setup, ExecuteDataSet calls da.Fill(ds). The Fill method executes the command in DataAdapter's SelectCommand property and populates ds with the query results. In the finally block, the method checks to make sure the objects it needs to dispose exists before calling the Dispose() method on them. It also checks to see if it is the owner of the connection before attempting to dispose / close the connection. Since this code exists in the finally block, it executes even if an exception occurs to ensure the objects are released. And lastly, it returns ds as the result of the function.

Listing 3 cont. – DataServiceBase class

```
//////////////////////////////////////////////////////////////////////
        // ExecuteNonQuery Methods

//////////////////////////////////////////////////////////////////////

        protected void ExecuteNonQuery (string procName,
            params IDataParameter[] procParams)
        {
            SqlCommand cmd;
            ExecuteNonQuery(out cmd, procName, procParams);
        }


        protected void ExecuteNonQuery(out SqlCommand cmd, string
procName,
            params IDataParameter[] procParams)
        {
            //Method variables
            SqlConnection cnx = null;
            cmd = null;  //Avoids "Use of unassigned variable" compiler
error

            try
            {
                //Setup command object
                cmd = new SqlCommand(procName);
                cmd.CommandType = CommandType.StoredProcedure;
                for (int index = 0; index < procParams.Length; index++)
                {
                    cmd.Parameters.Add(procParams[index]);
                }

                //Determine the transaction owner and process
accordingly
```

```
            if (_isOwner)
            {
                cnx = new SqlConnection(GetConnectionString());
                cmd.Connection = cnx;
                cnx.Open();
            }
            else
            {
                cmd.Connection = _txn.Connection;
                cmd.Transaction = _txn;
            }

            //Execute the command
            cmd.ExecuteNonQuery();
        }
        catch
        {
            throw;
        }
        finally
        {
            if (_isOwner)
            {
                cnx.Dispose(); //Implicitly calls cnx.Close()
            }
            if (cmd != null) cmd.Dispose();
        }
    }
```

Whenever you need to execute a statement that does not return query results, you use the ExecuteNonQuery methods. Once again, this implementation has two different version of the method that work with stored procedures, so if you want an ad-hoc SQL version then feel free to add it. The ExecuteNonQuery methods are almost identical to the ExecuteDataSet methods, so I will not cover them in too much detail. The main difference is that he ExecuteNonQuery method executes the SqlCommand object directly instead of setting up a SqlDataAdapter. Also note that ExecuteNonQuery does not have a return value because there is no query information to return.

Although the ExecuteNonQuery method does not return data via a DataSet, the stored procedures it executes often return output parameters and return values. As such, you are much more likely to use the out IDbCommand cmd parameter on the method than you would in the ExecuteDataSet method. Later on, when we get to the PersonDataService class, you will see this feature in action when we return auto-generated identity column values from a stored procedure.

Listing 3 cont. – DataServiceBase class

```
//////////////////////////////////////////////////////////////////
        // CreateParameter Methods

//////////////////////////////////////////////////////////////////
```

```
        protected SqlParameter CreateParameter(string paramName,
            SqlDbType paramType, object paramValue)
        {
            SqlParameter param = new SqlParameter(paramName, paramType);

            if (paramValue != DBNull.Value)
            {
                switch (paramType)
                {
                    case SqlDbType.VarChar:
                    case SqlDbType.NVarChar:
                    case SqlDbType.Char:
                    case SqlDbType.NChar:
                    case SqlDbType.Text:
                        paramValue =
CheckParamValue((string)paramValue);
                        break;
                    case SqlDbType.DateTime:
                        paramValue =
CheckParamValue((DateTime)paramValue);
                        break;
                    case SqlDbType.Int:
                        paramValue = CheckParamValue((int)paramValue);
                        break;
                    case SqlDbType.UniqueIdentifier:
                        paramValue =
CheckParamValue(GetGuid(paramValue));
                        break;
                    case SqlDbType.Bit:
                        if (paramValue is bool) paramValue =
                  (int)(((bool)paramValue ? 1 : 0));
                        if ((int)paramValue < 0 || (int)paramValue > 1)
paramValue = Constants.NullInt;
                        paramValue = CheckParamValue((int)paramValue);
                        break;
                    case SqlDbType.Float:
                        paramValue =
CheckParamValue(Convert.ToSingle(paramValue));
                        break;
                    case SqlDbType.Decimal:
                        paramValue =
CheckParamValue((decimal)paramValue);
                        break;
                }
            }
            param.Value = paramValue;
            return param;
        }

        protected IDataParameter CreateParameter(string paramName,
SqlDbType paramType,
            ParameterDirection direction)
        {
            IDataParameter returnVal = CreateParameter(paramName,
paramType, DBNull.Value);
            returnVal.Direction = direction;
            return returnVal;
```

```
        }

        protected IDataParameter CreateParameter(string paramName,
SqlDbType paramType,
            object paramValue, ParameterDirection direction)
        {
            IDataParameter returnVal = CreateParameter(paramName,
paramType, paramValue);
            returnVal.Direction = direction;
            return returnVal;
        }

        protected IDataParameter CreateParameter(string paramName,
SqlDbType paramType,
            object paramValue, int size)
        {
            IDataParameter returnVal = CreateParameter(paramName,
paramType, paramValue);
            ((SqlParameter)returnVal).Size = size;
            return returnVal;
        }

        protected IDataParameter CreateParameter(string paramName,
SqlDbType paramType,
            object paramValue, int size, ParameterDirection direction)
        {
            IDataParameter returnVal = CreateParameter(paramName,
paramType, paramValue);
            returnVal.Direction = direction;
            ((SqlParameter)returnVal).Size = size;
            return returnVal;
        }

        protected IDataParameter CreateParameter(string paramName,
SqlDbType paramType,
            object paramValue, int size, byte precision)
        {
            IDataParameter returnVal = CreateParameter(paramName,
paramType, paramValue);
            ((SqlParameter)returnVal).Size = size;
            ((SqlParameter)returnVal).Precision = precision;
            return returnVal;
        }

        protected IDataParameter CreateParameter(string paramName,
SqlDbType paramType,
            object paramValue, int size, byte precision,
ParameterDirection direction)
        {
            IDataParameter returnVal = CreateParameter(paramName,
paramType, paramValue);
            returnVal.Direction = direction;
            ((SqlParameter)returnVal).Size = size;
            ((SqlParameter)returnVal).Precision = precision;
            return returnVal;
        }
```

```
protected Guid GetGuid(object value)
{
    Guid returnVal = Constants.NullGuid;
    if (value is string)
    {
        returnVal = new Guid((string)value);
    }
    else if (value is Guid)
    {
        returnVal = (Guid)value;
    }
    return returnVal;
}
```

When you work with stored procedures you inevitably end up working with a lot of stored procedure parameters. Creating new SqlParameter objects is not always a simple task because you have to deal with setting the parameter name, direction, value, and sometimes even the size and precision. You also have to account for null values from your value-types because you need to pass in the DBNull.Value when you want the field to be null in the database. If you don't, you end up with a database full of crazy values (like -2147483648) instead of null values. So, the CreateParameter method exist to make creating stored procedure parameters much less painful.

The first CreateParameter method in the code listing is what does most of the work. It accepts three parameters: paramName is the name of the parameter, paramType is the parameter's database type, and paramValue is the value to assign to the parameter. The CreateParameter method begins by creating a new SqlParameter and passing in the parameter name and parameter type in the constructor. It then checks to see if the incoming paramValue is equal to DBNull.Value. If paramValue is NOT equal to DBNull.Value, then the method has to check the value against the appropriate null value type to see if the value should be treated as though it were null.

Notice, however, that the incoming paramValue is an object. This means that you don't know what type it is so you don't know what value-type to check it against. So we need a way to determine value's type before we can do the check. One option is to do direct type checking, but that requires writing a lot if statements that look messy. Another option is to determine the type and then use a switch statement on the type name, but that requires doing lots of comparisons against a string value, which just doesn't seem efficient. Instead, I chose to use the paramType in the switch statement because it's easy to map the SQL data type to the .NET data type. Inside each case statement, the method casts the parameter value to the appropriate type and passes it into the CheckParamValue method. This method evaluates the value against the appropriate null value and returns DBNull.Value if the value is marked as null. It the item is not marked as null it just returns the value unaltered.

Notice in the SqlDbType.UniqueIdentifer case statement that the method uses the GetGuid method to convert the incoming parameter value to a Guid instead of simply casting it. Sometimes Guid values are stored as strings and not as Guids, so the GetGuid method accounts for the possibility. If the value is a Guid, it simply returns the Guid as is.

If the value is a string, it creates a new Guid using the string. And if it's neither a Guid or a string, it simply returns a guid with a null value.

Also notice in the SqlDbType.Bit case statement that the method does a bit of extra processing. Boolean values in .NET can only have two values, true and false, making it impossible to setup a null value for a bool value type. If you want the ability to use null bool values, then you actually need to use an int to store the data because an int has the ability to store more than 2 values. This allows you to define 0 as false, 1 as true, and everything else as null. In the case statement, you can see that the method checks to see if the incoming value is a bool type and, if so, converts it into the appropriate int value. Otherwise, it assumes the incoming value is an int. On the next line the method ensures the int value is 0 or 1 and, if not, changes it to the null int value. It then runs the CheckParamValue against the int value to handle the value accordingly if it is marked as null.

Finally, the method sets the Value property on the SqlParameter object and returns it as the result of the method.

The rest of the CreateParameter overloads use the first one to setup the name, type, and value for a SqlParameter object, then set other miscellaneous properties on the object as needed. This gives you a set of CreateParameter methods that can easily handle a variation parameter properties.

Listing 3 cont. – DataServiceBase class

```
////////////////////////////////////////////////////////////////////
        // CheckParamValue Methods

////////////////////////////////////////////////////////////////////

        protected object CheckParamValue(Guid paramValue)
        {
            if (paramValue.Equals(Constants.NullGuid))
            {
                return DBNull.Value;
            }
            else
            {
                return paramValue;
            }
        }

        protected object CheckParamValue(string paramValue)
        {
            if (string.IsNullOrEmpty(paramValue))
            {
                return DBNull.Value;
            }
            else
            {
                return paramValue;
```

```
    }
}

protected object CheckParamValue(DateTime paramValue)
{
    if (paramValue.Equals(Constants.NullDateTime))
    {
        return DBNull.Value;
    }
    else
    {
        return paramValue;
    }
}

protected object CheckParamValue(double paramValue)
{
    if (paramValue.Equals(Constants.NullDouble))
    {
        return DBNull.Value;
    }
    else
    {
        return paramValue;
    }
}

protected object CheckParamValue(float paramValue)
{
    if (paramValue.Equals(Constants.NullFloat))
    {
        return DBNull.Value;
    }
    else
    {
        return paramValue;
    }
}

protected object CheckParamValue(Decimal paramValue)
{
    if (paramValue.Equals(Constants.NullDecimal))
    {
        return DBNull.Value;
    }
    else
    {
        return paramValue;
    }
}

protected object CheckParamValue(int paramValue)
{
    if (paramValue.Equals(Constants.NullInt))
    {
        return DBNull.Value;
    }
```

```
            else
            {
                return paramValue;
            }
        }

    } //class

} //namespace
```

And the last portion of the class consists of the CheckParamValue methods. These methods check a value against its null value and return DBNull.Value if the value is marked as null, or they return the value itself if the value is not marked as null. Each method contains an if statement that checks the parameter value against a null value from the Constants class, then returns the appropriate return value based on the outcome of the if statement.

Now that we have the DataServiceBase class and all of its helper methods, we can focus on building the PersonDataService class.

PersonDataService class – Demo.DataAccessLayer Assembly

Ther PersonDataService class contains all of the Data Access Methods for the person class. In our demo application we will be displaying a list of all the people in the system, displaying a single person, adding / editing a person, and deleting a person. So we need Data Access Methods for all of those scenarios. The database included with the demo application contains stored procedures to accomplish all of the items just mentioned. I am not going to cover the stored procedure code because it is not the focus of this article and it is not at all complicated. You can fire up SQL Server Management Studio and check out the code if you want. Listing 4 contains the code for the PersonDataService class.

Listing 4 – PersonDataService class

```
using System;
using System.Data;
using System.Data.SqlClient;
using Demo.Common;

namespace Demo.DataAccessLayer
{

    public class PersonDataService : DataServiceBase
    {


////////////////////////////////////////////////////////////////////
    // Constructors

////////////////////////////////////////////////////////////////////
```

```
        public PersonDataService() : base() { }


        public PersonDataService(IDbTransaction txn) : base(txn) { }



////////////////////////////////////////////////////////////////////
    // Data Access Methods
////////////////////////////////////////////////////////////////////

        public DataSet Person_GetAll()
        {
            return ExecuteDataSet("Person_GetAll", null);
        }


        public DataSet Person_GetByPersonID(int personID)
        {
            return ExecuteDataSet("Person_GetByPersonID",
                CreateParameter("@PersonID", SqlDbType.Int, personID));
        }


        public void Person_Delete(int personID)
        {
            ExecuteNonQuery("Person_Delete",
                CreateParameter("@PersonID", SqlDbType.Int, personID));
        }


        public void Person_Save(ref int personID, string nameFirst,
string nameLast, DateTime dob)
        {
            SqlCommand cmd;
            ExecuteNonQuery(out cmd, "Person_Save",
                CreateParameter("@PersonID", SqlDbType.Int, personID,
ParameterDirection.InputOutput),
                CreateParameter("@NameFirst", SqlDbType.NVarChar,
nameFirst),
                CreateParameter("@NameLast", SqlDbType.NVarChar,
nameLast),
                CreateParameter("@DOB", SqlDbType.DateTime, dob));
            personID = (int)cmd.Parameters["@PersonID"].Value;
            cmd.Dispose();
        }

    } //class

} //namespace
```

The PersonDataService class inherits the DataServiceBase class so it has to expose the same constructors as the DataServiceBase. Notice that there are two constructors for the class, both of which simply call down to the base class to handle the constructor logic. Such will be the case with all of your Data Service classes.

Then we move into the Data Access Methods. The first two methods return query results in the form of a DataSet object. The first is the Person_GetAll method which returns a DataSet containing a list of all the people in the system. This is an extremely simple method because the stored procedure it executes has no parameters. All you do is call the ExecuteDataSet method, passing in "Person_GetAll" as the stored procedure name and null as the list of parameters. Then you return the resulting DataSet from the ExecuteDataSet method.

People are uniquely identified in the system by their PersonID, an auto-generated int value. To acquire a single person from the system you use the Person_GetByPersonID method, which accepts a single int parameter named personID that specifies which person you are trying to retrieve. The method is still fairly simple, but the "Person_GetByPersonID" stored procedure requires you to pass it a stored procedure parameter named @PersonID to identiy which person you want it to retrieve. To do thise, you use the CreateParameter method inside the ExecuteDataSet to create a the @PersonID stored procedure parameter, set it's type to SqlDbType.Int, and assign if the value passed in via the personID method parameter. Then you return the resulting DataSet from the ExecuteDataSet method.

The Person_Delete method works just like the Person_GetByPersonID method, but it executes a non-query stored procedure named "Person_Delete" and does not need to return a DataSet. Since it doesn't need to return a DataSet, it uses the ExecuteNonQuery method instead of the ExecuteDataSet method.

And lastly we have the Person_Save method, which is responsible for adding and updating a person. A person object has four properties: PersonID is an int value that uniquely identifies the person, NameFirst and NameLast store the person's first and last name, and DOB stores the person's birth date. As mentioned before, when you want to save or update a business object, you have to pass all of the properties into the DAL, which is why the Person_Save method accepts all four of these parameters.

In the database, the PersonID is an identity column whose value is auto-generated when a new record is inserted. This complicates things a bit because it means we need to get that new value out of the database and into the business object in case the value needs to be referenced elsewhere. Part of the solution resides in the "Person_Save" stored procedure defined in the database. The stored procedure declares the @PersonID stored procedure parameter as an output parameter, and it updates the @PersonID stored procedure parameter to the appropriate value when a record is inserted. Since it's an output parameter, the new value is passed back out of the stored procedure so you can use place it back into the business object.

To do this, the Person_Save method declares the personID method parameter with the ref modifier. This means that the personID method parameter is passed by reference and updates to the value are passed back to the calling method. In the code, the method begins by declaring a SqlCommand named cmd. Notice that cmd is not initialized. The method

then calls ExecuteNonQuery and passes in cmd as an out parameter. It also sets up the necessary stored procedure parameters using the CreateParameter method. Note that when it creates the "@PersonID" stored procedure parameter that it sets the direction to be ParameterDirection.InputOutput. This lets the SqlCommand object know that it should pass the PersonID value into the stored procedure and to expect a value to come back out. Remember, this method can also be used to update an existing Person with an existing PersonID, so you need to set the direction to InputOuput and not just Output.

After the ExecuteNonQuery method fires, the cmd variable contains a SqlCommand object populated with the parameters you passed into the ExecuteNonQuery method. If any of those parameters are output parameters, then they contain the output values from the stored procedure. All you have to do is assign those output parameters to their appropriate method variables. So, after the ExecuteNonQuery method executes, the Person_Save method updates personID and sets it to the value in the "@PersonID" output parameter. Even if you are updating a person and not adding a new person, the "@PersonID" output parameter will have the correct PersonID value so it does not hurt to assign the "@PersonID" output parmater to personID each time. After acquiring output parameter values from the command, makes sure you dispose of the command appropriately. Next, it's on to the Business Tier.

DemoBaseObject class – Demo.Business Assembly

As its name implies, the DemoBaseObject is a base class for business objects in demo application. All of the methods in the DemoBaseObject revolve around mapping data into the business objects from a DataSet. Listing 5 contains the code for the DemoBaseObject and commentary follows thereafter.

Listing 5 – DemoBaseObject class

```
using System;
using System.Data;
using Demo.Common;

namespace Demo.Business
{
    public abstract class DemoBaseObject
    {


//////////////////////////////////////////////////////////////////////
        // MapData Methods

//////////////////////////////////////////////////////////////////////

        public virtual bool MapData(DataSet ds)
        {
            if (ds != null && ds.Tables.Count > 0 &&
ds.Tables[0].Rows.Count > 0)
            {
```

```
                return MapData(ds.Tables[0].Rows[0]);
            }
            else
            {
                return false;
            }
        }


        public virtual bool MapData(DataTable dt)
        {
            if (dt != null && dt.Rows.Count > 0)
            {
                return MapData(dt.Rows[0]);
            }
            else
            {
                return false;
            }
        }


        public virtual bool MapData(DataRow row)
        {
            //You can put common data mapping items here
            //(e.g. date record was created, last user to modify record,
etc)
            return true;
        }



////////////////////////////////////////////////////////////////////
        // Get Methods

////////////////////////////////////////////////////////////////////

        protected static int GetInt(DataRow row, string columnName)
        {
            return (row[columnName] != DBNull.Value) ?
                Convert.ToInt32(row[columnName]) : Constants.NullInt;
        }

        protected static DateTime GetDateTime(DataRow row, string
columnName)
        {
            return (row[columnName] != DBNull.Value) ?
                Convert.ToDateTime(row[columnName]) :
Constants.NullDateTime;
        }

        protected static Decimal GetDecimal(DataRow row, string
columnName)
        {
            return (row[columnName] != DBNull.Value) ?
                Convert.ToDecimal(row[columnName]) :
Constants.NullDecimal;
```

```
        }

        protected static bool GetBool(DataRow row, string columnName)
        {
            return (row[columnName] != DBNull.Value) ?
                Convert.ToBoolean(row[columnName]) : false;
        }

        protected static string GetString(DataRow row, string
columnName)
        {
            return (row[columnName] != DBNull.Value) ?
                Convert.ToString(row[columnName]) :
Constants.NullString;
        }

        protected static double GetDouble(DataRow row, string
columnName)
        {
            return (row[columnName] != DBNull.Value) ?
                Convert.ToDouble(row[columnName]) :
Constants.NullDouble;
        }

        protected static float GetFloat(DataRow row, string columnName)
        {
            return (row[columnName] != DBNull.Value) ?
                Convert.ToSingle(row[columnName]) : Constants.NullFloat;
        }

        protected static Guid GetGuid(DataRow row, string columnName)
        {
            return (row[columnName] != DBNull.Value) ?
                 (Guid)(row[columnName]) : Constants.NullGuid;
        }

        protected static long GetLong(DataRow row, string columnName)
        {
            return (row[columnName] != DBNull.Value) ?
                 (long)(row[columnName]) : Constants.NullLong;
        }

    } //class

} //namespace
```

Our object of choice for transferring data out of the DAL is the DataSet, but once you have a DataSet full of information you still have to pull data out of the DataSet and place it into your business objects. This process is known as data mapping because you are mapping data from the DAL into your business object. Before we get too far into the code, you need to be aware that a DataSet object may contain zero or more DataTable objects, and those DataTable objects may contain zero or more DataRow objects. Ultimately, the properties for your business objects reside in a DataRow object, so you need to find the appropriate DataRow and map the data from that DataRow into the business object.

Since there are three objects in a DataSet that you may encounter, there are three MapData methods in the DemoBaseObject class to handle each item accordingly. The first MapData method accepts an entire DataSet object. Now the method is really looking for a DataRow, but it's been passed a DataSet, so the method has to make some assumptions. It assumes that the data for the object is in the first DataRow of the first DataTable in the DataSet. In the code, it runs a check to make sure the DataSet is not null, that the first DataTable exists, and that the DataTable has at least one row of data. If it passes those checks, the method sends the first DataRow of the first DataTable to the MapData method that handles an individual DataRow. If not, it returns false to indicate the data was not mapped.

The second MapData method accepts a DataTable object and assumes the assumes the DataRow is the first row of data in the DataTable. The code checks to make sure the DataTable is not null and that it contains at least one row of data. If it passes those checks, the method sends the first DataRow in the DataTable to the MapData method that handles an individual DataRow. If not, it returns false indicating the data was not successfully mapped.

And the final MapData method, the one that all the other MapData methods ultimately call, accepts a DataRow. But if you look at the code for this MapData method then you will see that it just returns true and doesn't do anything else. Why? Remember, this is the base class from which all of your business objects derive. You need to override this MapData method in each of your business objects to populate whatever properties need to be populated from the DataRow. When we get to the Person class in a moment, you will see exactly how the mapping works.

One thing you may be asking at this is, wouldn't it be easier to just declare it an abstract method so your business objects are forced to override it? And the answer is yes, you could do it that way, but it's not as useful. Although it did not come up in the demo application, there are times when you have properties common to all of your business objects that need to be mapped. By declaring those mappings in the base class, you avoid having to recreate them in each individual business object. For example, let's say that your application needs to store the creation date, last modified date, and the last user who modified the record for each business object in your application. It's a whole lot easy to define those properties and the mappings in the base class then do recreate them who knows how many times in your business classes.

After the MapData methods there are nine "Get” methods that help in the data mapping process. Each method is responsible for returning a specific type from the DataRow, as well as converting a null value from the DataRow into the type's null value. This ensures that your application respects the concept of null coming out of the database. Every method accepts two parameters: a DataRow object containing information for your business object, and the column name to retrieve from that DataRow. The code then does a check against that column to see if it is null. If it is not null, then it casts the value into the appropriate types and returns it. If the column is null, then the method returns the null

value for the particular type. You'll see these methods in action when we look at data mapping in the Person class momentarily.

DemoBaseCollection class – Demo.Business Assembly

DemoBaseCollection serves as the base class for collections in your application. Like the DemoBaseObject class, the methods in the class center on data mapping, but now we have to deal with mapping a collection of data instead of just a single object. Listing 6 contains the code for the class.

Listing 6 - DemoBaseCollection

```
using System;
using System.Collections.Generic;
using System.Data;

namespace Demo.Business
{
    public abstract class DemoBaseCollection<T> : List<T> where T :
DemoBaseObject, new()
    {
        public bool MapObjects(DataSet ds)
        {
            if (ds != null && ds.Tables.Count > 0)
            {
                return MapObjects(ds.Tables[0]);
            }
            else
            {
                return false;
            }
        }

        public bool MapObjects(DataTable dt)
        {
            Clear();
            for (int i = 0; i<dt.Rows.Count; i++)
            {
                T obj = new T();
                obj.MapData(dt.Rows[i]);
                this.Add(obj);
            }
            return true;
        }

    } //class

} //namespace
```

The DemoBaseCollection is an abstract generic class that inherits its collection functionality from the generic List class.

Note: you can read some more about generics in the .NET Collection Management article by Amirthalingam Prasanna.

You can specify which type the collection represents by inheriting from the DemoBaseClass and specifying a type for <T>. Notice the where clause in the class definition. This specifies that the type you use as <T> must meet two criteria. It must be or derive from the DemoBaseObject class, and it has to have a zero parameter constructor. You'll see why we need to specify these criteria shortly. Our data mapping objective with the DemoBaseCollection is to locate a DataTable with zero or more rows of data, to map each DataRow in that DataTable to a new object, and to add the new object to the collection. So we only have to worry about two MapObjects methods, one that accepts a DataSet and one that accepts a DataTable.

The first MapObjects method accepts a DataSet object. The method begins by checking to make sure the DataSet is not null and that it contains at least one DataTable. If so, the method passes the first DataTable to the second MapObject method. Otherwise, it returns false to indicate there is no data to map.

The second MapObjects method begins by clearing any existing items out of the collection via the Clear method, which the DemoBaseCollection class inherits from the List class. It then enters a for loop that iterates over each DataRow object in the DataTable. Inside the for loop, the method creates a new object whose type corresponds to the generic type <T>. This is why the class definition contains the new() restriction on the generic type <T>, to ensure the object can be created inside the generic class with a zero-parameter constructor. It then passes the DataRow object into the object's MapData method. The object is known to have the MapData method because the generic type <T> also has a restriction that ensures the object derives from the DemoBaseObject class. Once the DataRow is mapped to the object, the method adds the new object to the class. It then returns true indicating the mapping was successful.

Now that we've got our base classes covered, let's move on to the Person class.

Person Class – Demo.Business Assembly

As mentioned before, the Person class contains four properties: PersonID, NameFirst, NameLast, and DOB. It also has methods to save a person, delete a person, and acquire a person by their PersonID. Listing 7 contains the code for the Person class.

Listing 7 – Person Class

```
using System;
using Demo.Common;
using System.Data;

namespace Demo.Business
{
```

```
    public class Person : DemoBaseObject
    {


////////////////////////////////////////////////////////////////////////
        // Fields
////////////////////////////////////////////////////////////////////////

        private int _personID = Constants.NullInt;
        private string _nameFirst = Constants.NullString;
        private string _nameLast = Constants.NullString;
        private DateTime _dob = Constants.NullDateTime;



////////////////////////////////////////////////////////////////////////
        // Properites
////////////////////////////////////////////////////////////////////////

        public int PersonID
        {
            get{return _personID;}
            set{_personID = value;}
        }

        public string NameFirst
        {
            get{return _nameFirst;}
            set{_nameFirst = value;}
        }

        public string NameLast
        {
            get{return _nameLast;}
            set{_nameLast = value;}
        }

        public DateTime DOB
        {
            get{return _dob;}
            set{_dob = value;}
        }



////////////////////////////////////////////////////////////////////////
        // Data Mapping and Data Access Methods
////////////////////////////////////////////////////////////////////////

        public override bool MapData(DataRow row)
        {
            PersonID = GetInt(row, "PersonID");
            NameFirst = GetString(row, "NameFirst");
            NameLast = GetString(row, "NameLast");
```

```
            DOB = GetDateTime(row, "DOB");
            return base.MapData(row);
        }

        public static Person GetByPersonID(int personID)
        {
            Person obj = new Person();
            DataSet ds = new
Demo.DataAccessLayer.PersonDataService().Person_GetByPersonID(personID);
            if (!obj.MapData(ds)) obj = null;
            return obj;
        }

        public static void Delete(int personID)
        {
            Delete(personID, null);
        }

        public static void Delete(int personID, IDbTransaction txn)
        {
            new
Demo.DataAccessLayer.PersonDataService(txn).Person_Delete(personID);
        }

        public void Delete()
        {
            Delete(PersonID);
        }

        public void Delete(IDbTransaction txn)
        {
            Delete(PersonID, txn);
        }

        public void Save()
        {
            Save(null);
        }

        public void Save(IDbTransaction txn)
        {
            new
Demo.DataAccessLayer.PersonDataService(txn).Person_Save(ref _personID,
_nameFirst, _nameLast, _dob);
        }

    } //class

} //namespace
```

I am not going to go into much detail on the first part of the Person class because it is relatively simple. You mainly need to know that the class inherits the DemoBaseObject class, so it has access to all of the data mapping method discussed previously. It also has four private fields to store properties values, and four public properties to expose those values.

The real code begins with the MapData override. This is where the Person class takes data from the DataRow and places it in the properties of the business object using the "Get" methods defined in the DemoBaseObject class. Remember, you pass in the DataRow object and the column name where the values resides, and the "Get" method returns the appropriately value for the item. Since you know the data type of the property you are wanting to populate, you can easily use the appropriate "Get" method when acquiring the value for that type. For example, the PersonID property is an int, so you use the GetInt method to acquire its value. NameFirst and NameLast are both strings, so you use the GetString method to acquire their values. And DOB is a DateTime, so you use GetDateTime to acquire its value. At the end of the override, make sure you call return base.MapData(row) to map any items you defined in the base class.

Next up is the static GetByPersonID method, which allows you to acquire a particular Person record based on their PersonID. It is the first method we are going to look at that calls the DAL to acquire query information. The method begins by creating a new Person object. On the next line, it acquires a DataSet containing the query results from the DAL. It does this by instantiating a new PersonDataService object and calling the Person_GetByPersonID method on that object, all on the same line. GetByPersonID then calls the Person object's MapData method inside an if statement. If the MapData method returns true, it means the data was successfully mapped to the object and the object can be returned as the result of the method. If the MapData method returns false, then the method sets the Person object to null and returns that as the result of the method to indicate that the person was not found.

The first two Delete methods are static methods that allows you delete a person's record given their PersonID. When you delete a person you are executing a statement and not a query, so you do not have to worry about mapping data or returning an object from the method. But you do have to worry about whether or not you want to execute the method inside a transaction. That's why there are two versions of the static Delete method, one for deleting without a transaction, and one for deleting within a transaction. The version for deleting without a transaction simply calls the transactional version but passes in a null transaction. There are also non-static versions of the Delete method that you can call from a Person object. These simply call the static version of the Delete method and pass in the object's PersonID to identify which record to delete.

And lastly, you have the Save methods, which allows you to save the properties in the Person object to the database. As was the case with the Delete method, there are two versions of the method to allow saving a Person with or without a transaction. The method simply creates a new PersonDataService and passes in the appropriate field variables to the Person_Save method. Notice that the _personID field is passed with the ref modifier, which allows it to be updated inside the DAL. It needs to be updateable because the PersonID is auto-assigned by the database when a record is inserted.

As you look over the Person class, notice that it does not contain any database-specific code. Any time the class needs to access data, it hits the DAL.

PersonCollection Class – Demo.Business Assembly

We will be dealing with lists of People, so we need a PersonCollection to help store and manage that list. Compared to the other classes we've been looking at, this one is a breeze. Listing 8 has the code for the PersonCollection class.

Listing 8 – Person Class

```
using System;
using Demo.DataAccessLayer;

namespace Demo.Business
{
    public class PersonCollection : DemoBaseCollection<Person>
    {
        public static PersonCollection GetAll()
        {
            PersonCollection obj = new PersonCollection();
            obj.MapObjects(new PersonDataService().Person_GetAll());
            return obj;
        }

    }
}
```

PersonCollection inherits its collection functionality from the generic DemoBaseCollection class (which in terms inherits it form the generic List class). Since you want the collection to be strongly-typed for the Person class, you pass the Person type in the class declaration. Since person has zero-paramater constructor and derives from DemoBaseObject, so it will pass the type requirements imposed by the where clause in the DemoBaseCollection. This class only has one method, a static method that returns a PersonCollection populated with all of the Person records in the system. It does this by first creating a new PersonCollection object named obj. It then creates a new PersonDataService object, calls the Person_GetAll method to acquire a DataSet containing the Person records, and passes that DataSet into the MapObjects method of the collection. And it does it all on the same line of code for the sake of brevity. Then it returns the object as the result of the method.

Notice that we do not check the result of the MapObjects method to see if comes back true or false. If you want to return a null object if the mapping fails, then you may do so. I prefer to let the collection come back empty instead of null, which is what happens if there is no data present.

Now we have all of our business objects, so all that's left is to use them in the presentation layer.

Displaying and Deleting People – PersonShowAll.aspx

On the PersonShowAll.aspx page, there is a GridView control named gridPeople that allows you to see a list of people in the system. There are links next to the Person to jump to the PersonEdit.aspx page, and checkboxes allowing you to select a series of people to delete. In the code behind, setting up the gridPeople GridView to display the list of people is extremely easy. Listing 9 shows the code required to do so.

Listing 9 – Setting up the gridPeople GridView Control

```
    protected void Page_Load(object sender, EventArgs e)
    {
        gridPeople.DataSource = PersonCollection.GetAll();
        gridPeople.DataBind();
    }
```

All you have to do is call the static GetAll method on the PersonCollection, assign the resulting value to the GridView's DataSource property, then DataBind the grid.

Deleting users is also very simple, but it does require a bit of background. The GridView displays a series of checkboxes, all of which are named chkSelectedItems. The value of each checkbox is set to the PersonID of the Person it represents. When an HTML form contains multiple checkboxes with the same name, it takes the values of all the checkboxes you selected and places them into a comma separated list. This is very handy when you want to get back a list of items that were selected. So, you can select people on the PersonShowAll.aspx page then click on the lnkDeleteSelected link button to delete the selected users. Listing 10 shows the code for the deletion process.

Listing 10 – Deleting Users

```
protected void lnkDeleteSelected_Click(object sender, EventArgs e)
{
    if (!string.IsNullOrEmpty(Request.Form["chkSelectedItems"]))
    {
        string[] selectedItems =
Request.Form["chkSelectedItems"].Split(',');
        for (int i = 0; i < selectedItems.Length; i++)
        {
            try
            {
                Person.Delete(int.Parse(selectedItems[i]));
            }
            catch
            {
            }
        }
        gridPeople.DataSource = PersonCollection.GetAll();
        gridPeople.DataBind();
    }
}
```

This piece of code begins by checking the Form collection of the Request object to see if "chkSelectedItems" has a list of values. If so, it splits the string using a comma as the delimiter and stores the string values in an array named selectedItems. It then iterates through each item in the array, casts the item to an integer, and sends that integer to the static Person.Delete method. This deletes all of the selected users. It then rebinds the grid so the deleted users are not displayed. That sums up the PersonShowAll.aspx page, so we'll move on to the PersonEdit.aspx page next.

Adding and Editing People – PersonEdit.aspx

You can add and edit people from the PersonEdit.aspx. The page accepts a query string parameter named PersonID that identifies the person to edit. If the PersonID is missing, zero, or invalid, the page assumes you are adding a new person. There are three text fields on the page that allow you to edit the Person object's first name, last name, and date of birth. Listing 11 contains the code that loads the appropriate Person object and displays it's properties in the appropriate fields on the page.

Listing 11 – Loading and Displaying a Person

```
private Person _person;

protected void Page_Load(object sender, EventArgs e)
{
    //Acquire the PersonID from the querystring
    int personID = 0;
    if (Request.QueryString["PersonID"] != null)
    {
        try
        {
            personID = int.Parse(Request.QueryString["PersonID"]);
        }
        catch
        {
        }
    }

    //Attempt to get the person if there is a PersonID
    if(personID > 0) _person = Person.GetByPersonID(personID);

    //Populate the form if this is the first request
    if(!IsPostBack)
    {
        //Only populate the form if the person object is valid
        if (_person != null)
        {
            lblPersonID.Text = _person.PersonID.ToString();
            txtNameFirst.Text = _person.NameFirst;
            txtNameLast.Text = _person.NameLast;
            if (_person.DOB != Constants.NullDateTime)
            {
                txtDOB.Text = _person.DOB.ToString("M/d/yyyy");
            }
```

```
        }
        else
        {
            //Display a message indicating that you are adding a person
            lblPersonID.Text = "New Person";
        }
    }
}
```

The first part of the code acquires the PersonID from the query string. Notice if there is no PersonID query string value, or if there is an error parsing the value from the query string, PersonID defaults to 0. The method then checks to see if personID is greater than zero and, if so, calls Person.GetByPersonID to load the appropriate Person object. If the PersonID passed into the method does not exist, then Person.GetPersonByID returns null.

Next, the method checks to see if this is the first time the page has loaded to determine if it needs to populate the form fields with data from the Person object. If so, it checks to see if the Person object exists and, assuming it does, loads the appropriate properties into the appropriate fields. If not, it displays a message in the lblPersonID label telling the user that they are adding a new person.

After the user has updated the fields on the page, they click on the cmdSave button to save their changes. Listing 12 contains the code to repopulate the object from the form fields and save the object.

Listing 12 – Saving Changes to the Person

```
protected void cmdSave_Click(object sender, EventArgs e)
{
    if (Page.IsValid)
    {
        if (_person == null) _person = new Person();
        _person.NameFirst = txtNameFirst.Text;
        _person.NameLast = txtNameLast.Text;
        if (!string.IsNullOrEmpty(txtDOB.Text))
        {
            _person.DOB = Convert.ToDateTime(txtDOB.Text);
        }
        else
        {
            _person.DOB = Constants.NullDateTime;
        }

        _person.Save();

        //At this point _person.PersonID has been updated with
        //it's new identity value (if the person was just added)

        Response.Redirect("PersonShowAll.aspx");

    }
}
```

This code listing begins by checking the Page.IsValid property to ensure the incoming form fields passed validation. If all the data is valid, the method checks to see if the _person variable points to a valid object. Remember, _person is populated in the Page_Load event if there is a valid PersonID in the query string. If _person is null, the method assumes you are adding a new person, so it creates a new Person object and assigns it to the _person variable. The method then sets the NameFirst and NameLast properties on the Person object to the values coming from the txtNameFirst and txtNameLast textboxes on the form. After that, it checks to see if the txtDOB textbox has any text. If so, it converts the text into a DateTime value and assigns it to the DOB property on the Person object. If the txtDOB textbox does not have any text, the method sets the DOB property on the Person object to the null DateTime value.

Once the properties on the Person object have been updated, the method calls _person.Save to save the properties to the data store. Although we do not use the PersonID property in this code listing, know that you could use it after calling Save and it will contain the auto-generated PersonID value from the database if you just added the person.

Working with Transactions

There are times when you need to use a transaction to ensure changes to multiple business objects are either committed to the database together, or not at all. You can acquire an IDbTransaction object using the DataServiceBase.BeginTransaction method. You need to store that transaction in a variable so you can pass it to the various data-related methods on your business object, and so you can commit or rollback the transaction in your code. Listing 13 shows a hypothetical situation that uses a transaction.

Listing 13 – Using a Transaction

```
Person personA = Person.GetByPersonID(1);
Person personB = Person.GetByPersonID(2);
Person personC = Person.GetByPersonID(3);
Invoice invoiceA = Invoice.GetByInvoiceID(99);  //hypothetical business
object

// … code that updates the objects

IDbTransaction txn = DataServiceBase.BeginTransaction();
try
{
    personA.Save(txn);
    personB.Save(txn);
    personC.Delete(txn);
    invoiceA.Save(txn);
    txn.Commit();
}
catch
{
    //An error occured so roll the transaction back
```

```
    txn.Rollback();
}
```

Listing 13 begins by creating and populating four variables: three Person variables and one Invoice variable (assume Invoice is a business object like Person). After running some code that updates the values, you need to save personA, personB, invoiceA, and delete personC. You want the updates and deletions to succeed or fail together, so you begin a transaction by calling DataServiceBase.BeingTransaction() and store the resulting IDbTransaction object in the txn variable. You then enter a try/catch block and call the various save and delete methods on your business objects. If there are no errors, then your code calls txn.Commit() to commit the changes together. If there is an error saving or deleting an item, the exception forces the code into the catch portion of the try/catch block where the transaction is rolled back.

Conclusion

Whew! There were a lot of concepts and code to cover in this article, but I wanted to make sure you could get a good feel not only for the concepts but also the implementation details involved in the process. At this point, you should have a good design-level understanding of the DAL, where it fits into the overall architecture of an application, how to build the individual Data Services and Data Access methods that make up the DAL, and how to use the DAL in your business object code. You also have a full blown DAL implementation to refer back to in case you need any of the specifics. Have fun with it!