ADO.NET Data Wrapper Classes

When developers think of how to access data, many use the Entity Framework (EF), Dapper, NHibernate, or some other object-relational mapper (ORM). Each of these ORMs use ADO.NET to submit their SQL queries to the backend database. So, why do we use ORMs instead of just using ADO.NET directly? Simply put, ORMs allow you to write less code. If each of these ORMs are simply wrappers around ADO.NET, can't you write your own wrapper to cut down the amount of code you need to write? Absolutely! This article describes a set of wrapper classes to make it simpler to work with ADO.NET. This article does not go into every line of code in the wrapper, it is intended as an overview of the functionality.

Problems with ORMs

While there is nothing wrong with the Entity Framework (EF) or most object-relational mappers (ORMs), most of them are slower than ADO.NET. This is not to say they are slow to the point where you users would notice. At least, not in most cases, but they all have the potential to have performance issues. When using EF, for example, you often see performance issues when using LINQ to perform complicated joins. Complicated queries belong in a stored procedure, not in C# code. There is no way to optimize a complicated query using LINQ. However, your DBA can do wonders with complicated queries located in a stored procedure.

Another reason to use ADO.NET is Microsoft does not modify the API to the ADO.NET classes anymore. However, EF, and the other ORMs are constantly changing and evolving, and many times introduce breaking changes. This means when you upgrade from one version of .NET to another, your code may no longer compile.

When using ORMs you typically need to learn another language such as LINQ or learn how to configure the ORM using configuration files or C# classes. Sometimes there are special tools for regenerating classes if you modify your schema. You must learn how to use these new tools. And, many times these tools are upgraded, and you must re-learn how to use them all over again.

Some ORMs make it difficult to work with stored procedures. The Entity Framework has notoriously been bad at calling stored procedures, though it has improved in the later versions.

Advantages to Working with ORMs

While there are issues when working with any tool, there are also advantages. For ORMs the major advantage is you typically end up writing a lot less C# code. Part of the way they do this is automatic mapping of columns in a table to properties in your class.

Another big advantage is they have some code generation tools that can read a database and generate your set of classes to interact with the tables. Some even go the other way and read your C# classes and generate a database schema.

Advantage of Using ADO.NET Wrapper Classes

The wrapper classes described in this article are not intended to be an ORM. They do have many advantages, however.

1. You write much less C# code than normal ADO.NET code
2. Performance is better since you are using ADO.NET directly
3. You don't have to learn any new tools or configuration
4. Easy to work with any .NET data provider

The following are the features the ADO.NET data wrapper classes provide.

1. Use dynamic SQL or stored procedures easily
2. Build a DataReader and automatically populate a List<T> collection
3. Build a DataTable and automatically populate a List<T> collection
4. Use [Column] attribute to populate data from DataReader or DataTable
5. Retrieve a scalar value
6. Execute an action query
7. Retrieve IDENTITY value using dynamic SQL or stored procedure
8. Handles multiple result sets
9. Handles transactions
10. Validate data using data annotations and your own custom code
11. Creates an exception object with the SQL statement submitted, a redacted connection string, command parameters and values, stack trace, machine name, and more items
12. Support for any .NET database provider such as SQL Server and Oracle

Overview of ADO.NET Wrapper Classes

The following classes are in the sample application that comes with this article.

Class Name	Description
Manager Classes	
DataManagerBase	A base class located in a Common.DataLayer class library project. This class is responsible for submitting dynamic SQL or stored procedure calls. Parameters can (optionally) be passed regardless of the method used to query the database. This class inherits from the CommonBase class.
SqlServerDataManagerBase	This class inherits from the DataManagerBase class and overrides specific methods to provide concrete implementations of SQL Server specific classes.
AppDataManagerBase	This class inherits from the SqlServerDataManagerBase class. You use this class if you have any standard input and/or output parameters you want to add to every stored procedure call. Otherwise, it simply calls the methods within the base class. This class' constructor passes the name of the connection string element to the DataManagerBase class to use to retrieve the connection string from the config file. Or, you can pass in a connection string.
ProductManager	This class inherits from AppDataManagerBase class so you can interact with the AdventureWorksLT database. This class contains methods to perform selecting, counting, searching, inserting, updating, validating and deleting data contained within the Product table using dynamic SQL. This class uses a SqlDataReader to retrieve all data.
ProductSPManager	This class is the same as the ProductManager class, except it submits queries to the database using stored procedures.
ProductDataSetManager	This class is the same as the ProductManager class except it uses a SqlDataAdapter to fill a DataSet instead of using a SqlDataReader for retrieving data.
ProductSPDataSetManager	This class is the same as the ProductSPManager class except it uses a SqlDataAdapter to fill a DataSet instead of using a SqlDataReader for retrieving data.
ProductCategoryManager	This class uses a SqlDataReader to retrieve data. The ProductCategory entity class it uses to build a collection of product category objects uses the [Column] attribute to map a field name to a specific property when using reflection to automatically populate the list of record.
Entity Classes	

EntityBase	This class has no methods or properties but provides us with a base class in case you wish to add a set of properties and/or methods for all your Entity classes.
SqlServerEntityBase	This class inherits from the EntityBase class and adds a RETURN_VALUE property. This property is a standard property returned from all SQL Server stored procedures. By placing the property here you don't have to add it to all your Entity classes.
AppEntityBase	This class inherits from the SqlServerEntityBase class. This class has no methods or properties but provides us with a base class for this application in case you have a set of properties and/or methods for all entity classes within this specific application.
Product	This class inherits from the AppEntityBase class and has one property for each column in the Product table.
ProductCategory	This class inherits from the AppEntityBase class and has one property for each column in the ProductCategory table.
ProductSearch	Provides a class to hold data the user inputs in order to search for specific rows in the Product table.
View Model Classes	
ViewModelBase	A base view model class for any view model located in the Common.Library class library project.
AppViewModelBase	A base view model class specific for this sample located in the DataWrapper.Samples.AppLayer class library project. This class inherits from the ViewModelBase class.
ProductViewModel	A view model class for working with product data located in the DataWrapper.Samples.ViewModelLayer class library project. This class inherits from the AppViewModelBase class. This class calls the ProductManager class to perform all database operations.
ProductCategoryViewModel	A view model class that calls the ProductCategoryManager class to perform all database operations.
ProductDataSetViewModel	A view model class that calls the ProductDataSetManager class to perform all database operations.
ProductSPViewModel	A view model class that calls the ProductSPManager class to perform all database operations.
ProductSPDataSetViewModel	A view model class that calls the ProductSPDataSetManager class to perform all database operations.

In the sample for this article, you are working with a Product table. This means you need a ProductManager class to work with this table. For each table in your database, you need a "manager" class. Each of these manager classes inherits from the AppDataManager class as shown in Figure 1.

Create a single Entity class for each table you wish to interact with, or with any stored procedure that you wish to return data from. Instead of adding one Entity class for each stored procedure, you may reuse the Entity classes for your tables.

Feel free to add additional properties as necessary to support joined data being returned from stored procedures.

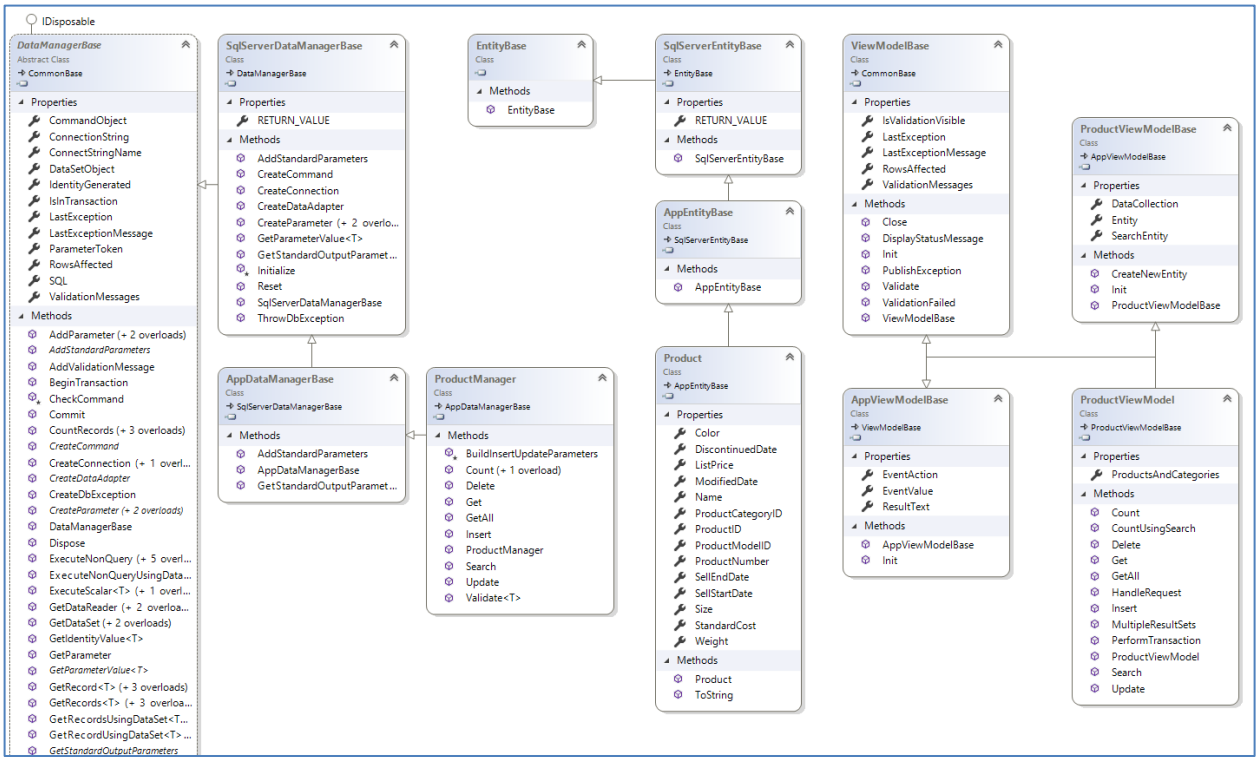

Figure 1: A diagram of the classes used to query a database using dynamic SQL or stored procedures.

The Product Entity Class

While you are not using the Entity Framework, you should still add the appropriate data annotations to match up the columns in the table with properties in your entity class. In addition, add data annotations to handle validation. Below is the Product class with the data annotations so each field is marked with the appropriate attributes to help with validation.

```
[Table("Product", Schema ="SalesLT")]
public partial class Product : EntityBase
{
  [Required]
  [Key]
  [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
  public int ProductID { get; set; }

  [Required]
  public string Name { get; set; }

  [Required]
  public string ProductNumber { get; set; }
  public string Color { get; set; }

  [Required]
  public decimal StandardCost { get; set; }

  [Required]
  public decimal ListPrice { get; set; }

  public string Size { get; set; }
  public decimal? Weight { get; set; }
  public int? ProductCategoryID { get; set; }
  public int? ProductModelID { get; set; }

  [Required]
  public DateTime SellStartDate { get; set; }
  public DateTime? SellEndDate { get; set; }
  public DateTime? DiscontinuedDate { get; set; }

  [Required]
  public DateTime ModifiedDate { get; set; }

  public byte[] ThumbNailPhoto { get; set; }
  public string ThumbnailPhotoFileName { get; set; }
  public Guid rowguid { get; set; }

  public override string ToString()
  {
    return ProductNumber + " (" + ProductID.ToString() + ")";
  }
}
```

The ProductCategory Entity Class

This class uses the [Column] attribute to map the field name in the ProductCategory table to a different property name in this class.

```
public partial class ProductCategory : AppEntityBase
{
  /// <summary>
  /// Get/Set ProductCategoryID
  /// </summary>
  [Required]
  [Column("ProductCategoryID")]
  public int Id { get; set; }

  /// <summary>
  /// Get/Set Name
  /// </summary>
  [Required]
  [Column("Name")]
  public string CategoryName { get; set; }

  /// <summary>
  /// Get/Set ModifiedDate
  /// </summary>
  [Required]
  public DateTime ModifiedDate { get; set; }

  public override string ToString()
  {
    return CategoryName + " (" + Id.ToString() + ")";
  }
}
```

The Product Manager Class

This section presents the ProductManager class. This class is used to call the submit the appropriate SQL to the AdventureWorksLT database. This class helps you perform CRUD operations against the Product table. For each table you have in your database, you generally create a corresponding "manager" class to retrieve data from and modify data within each table.

You can either create additional "manager" classes to call stored procedures that perform joins between tables or place these calls within existing table "manager" classes.

Get All Records

The GetAll() method in the ProductManager class is used to retrieve all records from the Product table.

```
public virtual List<Product> GetAll()
{
  // Get all rows from SalesLT.Product
  return GetRecords<Product>("SELECT * FROM SalesLT.Product");
}
```

If you open the DataManagerBase class and look at the GetRecords() method you see it calls the Reset() method to reset the Command object, reset the SQL property and a few other properties to initial start values. Fill in the *SQL* property with the SQL statement you wish to submit to the database. Utilize the using() block around the call to the GetDataReader() method from the DataManagerBase class. This method returns an instance of a .NET IDataReader object. Pass this data reader to the ToList<T>() method along with the type of object you wish to create a list of entity objects of. This ToList() method, located in the DataManagerBase class uses reflection to create a list of entity objects.

The ToList() method is very efficient, and actually works faster than the Entity Framework since there is not as much overhead required by EF.

Calling the GetAll() Method

Calling the methods in the ProductManager class are generally very simple. For example, to call the GetAll() method, you write code like the following:

```
public void GetAll()
{
  using (ProductManager mgr = new ProductManager()) {
    try {
      var list = mgr.GetAll();
      var rowsAffected = mgr.RowsAffected;
    }
    catch (Exception ex) {
      PublishException(ex);
    }
  }
}
```

Notice the use of the using() block. This is done so the command and connection objects can be disposed of correctly.

Get All Records using Stored Procedure

If you want to use a stored procedure to retrieve all records, this method requires just a few changes. Here is the GetAll() method from the ProductSPManager class.

```
public virtual List<Product> GetAll()
{
  // Get all rows from SalesLT.Product
  return GetRecords<Product>("SalesLT.Product_GetAll",
    CommandType.StoredProcedure);
}
```

Pass in the CommandType to the GetAllRecords() method and specify that you are putting the name of a stored procedure in the *SQL* property. If you look in the GetAllRecords() method in the DataManagerBase class you see a call to the AddStandardParameters() method. This method is overridden in the AppDataManagerBase class and is used if you have any standard parameters you normally add to your stored procedure for this application. There is also a call to the GetStandardOutputParameters() method in case you have any standard OUTPUT parameters in your stored procedures.

Below is the stored procedure text added to the AdventureWorksLT database.

```
CREATE PROCEDURE SalesLT.Product_GetAll
AS
BEGIN
  SELECT
[ProductID],[Name],[ProductNumber],[Color],[StandardCost],[ListPrice
],[Size],[Weight],[ProductCategoryID],[ProductModelID],[SellStartDat
e],[SellEndDate],[DiscontinuedDate],[ModifiedDate]
  FROM SalesLT.Product;
END
```

Get a Single Product

If you wish to retrieve a single product value, you need to add a WHERE clause and a parameter for the primary key as shown in the following Get(int) method.

```
public virtual Product Get(int productId)
{
  // Create SQL to SELECT FROM SalesLT.Product
  SQL = "SELECT * FROM SalesLT.Product
         WHERE ProductID = @ProductID";

  // Create parameters for counting
  var parameters = new List<IDbDataParameter> {
    // Add parameters for CommandObject
    CreateParameter("ProductID", (object)productId, false)
  };

  return GetRecord<Product>(SQL, parameters.ToArray());
}
```

The code for the Get(int) method is very similar to the GetAll() method you just wrote. The difference is a parameter named @ProductID is passed within the *SQL* property. If you have a parameter(s) you need to call the CreateParameter() method for each parameter in your SQL statement.

Get a Single Product using a Stored Procedure

In the AdventureWorksLT database, I added a stored procedure to retrieve a single product from the SalesLT.Product table. This stored procedure has a single parameter, @ProductID, of the type int. The ProductID field is the primary key for the Product table. This stored procedure, named SalesLT.Product_Get looks like the following:

```
CREATE PROCEDURE [SalesLT].[Product_Get]
 @ProductID int
AS
BEGIN
 SELECT *
 FROM SalesLT.Product
 WHERE ProductID = @ProductID;
END
```

To call this stored procedure, a method named Get(int productId) is created in the ProductSPManager. This method is shown below.

```
public virtual Product Get(int productId)
{
  // Create SQL to SELECT FROM SalesLT.Product
  SQL = "SalesLT.Product_Get";

  // Create parameters for counting
  var parameters = new List<IDbDataParameter> {
    // Add parameters for CommandObject
    CreateParameter("ProductID", (object)productId, false)
  };

  return GetRecord<Product>(SQL, CommandType.StoredProcedure,
             parameters.ToArray());
}
```

This method is not too different from the Get(int) method in the ProductManager class. The only difference is the call to the GetRecord() method passes in the CommandType of StoredProcedure.

GetAll with Output Parameter

If you have a stored procedure that returns a result set and has an OUTPUT parameter, you need to perform a couple additional steps. Below is the Product_GetAllWithOutput stored procedure.

```
CREATE PROCEDURE [SalesLT].[Product_GetAllWithOutput]
 @Test nvarchar(10) OUTPUT
AS
BEGIN
 SELECT *
 FROM SalesLT.Product;

 SELECT @Test = 'Hello';
END
```

To call this stored procedure, a method named GetAllWithOutputParameter() is created.

```
public virtual List<Product> GetAllWithOutputParameter()
{
  List<Product> ret;

  // Create SQL to call stored procedure
  SQL = "SalesLT.Product_GetAllWithOutput";

  // Add Ouput Parameter
  var parameters = new List<IDbDataParameter> {
    // Add parameters for CommandObject
    CreateParameter("@Test", "", false, DbType.String,
      10, ParameterDirection.Output)
  };

  // Get all rows from SalesLT.Product
  ret = GetRecords<Product>(SQL, CommandType.StoredProcedure,
                            parameters.ToArray());

  // Get @Test Output Parameter
  string test = base.GetParameterValue<string>("@Test", "");
  Console.WriteLine(test);

  return ret;
}
```

In the code above, prior to calling the stored procedure, add the output parameter. After you make the call to the retrieve the result set, use the GetParameterValue() method to retrieve the output parameter from the stored procedure call.

Search for Products

Sometimes you wish to ask a user to input one or more parameters to search for data in your table. In this sample the user can enter the name of a product (or a partial name), a product number (or partial product number), and a range of a beginning and ending costs to search for. To hold this data for searching, create a ProductSearch class as shown below.

```
public class ProductSearch
{
  public string Name { get; set; }
  public string ProductNumber { get; set; }
  public decimal? BeginningCost { get; set; }
  public decimal? EndingCost { get; set; }
}
```

To make the call to the search for the data, create a method named Search()as shown below.

```
public virtual List<Product> Search(ProductSearch search)
{
  // Create SQL to SELECT FROM SalesLT.Product
  SQL = "SELECT * FROM SalesLT.Product";
  SQL += " WHERE (@Name IS NULL OR NAME LIKE @Name + '%')";
  SQL += " AND (@ProductNumber IS NULL OR
                ProductNumber LIKE @ProductNumber + '%')";
  SQL += " AND (@BeginningCost IS NULL OR
                StandardCost >= @BeginningCost)";
  SQL += " AND (@EndingCost IS NULL OR
                StandardCost <= @EndingCost)";

  // Create parameters for searching
  var parameters = new List<IDbDataParameter> {
    // Add parameters for CommandObject
    CreateParameter("Name",
      (object)search.Name ?? DBNull.Value, true),
    CreateParameter("ProductNumber",
      (object)search.ProductNumber ?? DBNull.Value, true),
    CreateParameter("BeginningCost",
      (object)search.BeginningCost ?? DBNull.Value, true),
    CreateParameter("EndingCost",
      (object)search.EndingCost ?? DBNull.Value, true)
  };

  return GetRecords<Product>(SQL, CommandType.Text,
                             parameters.ToArray());
}
```

In the CreateParameter() method, the second parameter should be either the data from the ProductSearch class, or a DBNull value if the data in the ProductSearch

object is a null. The third parameter to CreateParameter() specifies if the parameter is allowed to be a null value.

Count Products

Some other functionality you might want is the ability to count all rows within a table, or count rows based on some criteria. There are two Count() methods created in the ProductManager class. Create a Count() method to count all rows in the Product table.

```
public virtual int Count()
{
  // Count all records in Product table
  return CountRecords("SELECT Count(*) FROM SalesLT.Product");
}
```

Use the CountRecords() method in the DataManagerBase class to return the single value returned from the SQL statement.

Create a Count(ProductSearch) method to which you pass an instance of the ProductSearch class. This method counts rows according to the WHERE clause used.

```
public virtual int Count(ProductSearch search)
{
  // Create SQL to count rows
  SQL = "SELECT Count(*) FROM SalesLT.Product";
  SQL += " WHERE (@Name IS NULL OR NAME LIKE @Name + '%')";
  SQL += " AND (@ProductNumber IS NULL OR
                ProductNumber LIKE @ProductNumber + '%')";
  SQL += " AND (@BeginningCost IS NULL OR
                StandardCost >= @BeginningCost)";
  SQL += " AND (@EndingCost IS NULL OR
                StandardCost <= @EndingCost)";

  // Create parameters for counting
  var parameters = new List<IDbDataParameter> {
    // Add parameters for CommandObject
    CreateParameter("Name", (object)search.Name ?? DBNull.Value,
      true),
    CreateParameter("ProductNumber", (object)search.ProductNumber
      ?? DBNull.Value, true),
    CreateParameter("BeginningCost", (object)search.BeginningCost
      ?? DBNull.Value, true),
    CreateParameter("EndingCost", (object)search.EndingCost
      ?? DBNull.Value, true)
  };

  return CountRecords(SQL, CommandType.Text, parameters.ToArray());
}
```

Insert a Product

To insert data into the Product table, you need an INSERT statement. This can be either in dynamic SQL or a stored procedure. Below is an Insert() method in the ProductManager class that inserts a new product row.

```
public virtual int Insert(Product entity)
{
  // Reset all properties
  Reset();

  // Attempt to validate the data,
  // a ValidationException is thrown if validation rules fail
  Validate<Product>(entity);

  // Create SQL to INSERT INTO SalesLT.Product using dynamic SQL
  SQL = "INSERT INTO SalesLT.Product(Name, ProductNumber,
            Color, StandardCost, ListPrice, Size, Weight,
            ProductCategoryID, ProductModelID, SellStartDate,
            SellEndDate, DiscontinuedDate, ModifiedDate) ";
  SQL += " VALUES(@Name, @ProductNumber, @Color, @StandardCost,
           @ListPrice, @Size, @Weight, @ProductCategoryID,
           @ProductModelID, @SellStartDate, @SellEndDate,
           @DiscontinuedDate, @ModifiedDate)";

  // Create standard insert parameters
  BuildInsertUpdateParameters(entity);

  // Execute Query and retrieve the IDENTITY
  RowsAffected = ExecuteNonQuery(true);

  // Get the ProductID generated from the IDENTITY
  entity.ProductID = GetIdentityValue<int>(-1);

  return RowsAffected;
}
```

This method first validates the product data to ensure it can be inserted into the table by calling the Validate() method. This method is described later in this article. If the validation passes, set the *SQL* property to the name of the stored procedure and add all the parameters to insert, plus the OUTPUT parameter. A method named BuildInsertUpdateParameters(), described in the next section, is called to add all the appropriate parameters to the command object.

Call the ExecuteNonQuery() method passing in a true value to specify that you want to retrieve the IDENTITY value generated from this INSERT statement. The *IdentityGenerated* property is automatically filled in when you pass a true value to the ExecuteNonQuery() method. Retrieve the *IdentityGenerated* property using the GetIdentityValue() method.

Build Insert/Update Parameters Method

When you are inserting or updating the Product table, you most likely need to pass the same set of parameters to either stored procedure. Create one method, BuildInsertUpdateParameters(), with the parameters that are in common between the two stored procedures.

```
protected virtual void BuildInsertUpdateParameters(Product entity)
{
  // Add parameters to CommandObject
  AddParameter("Name", (object)entity.Name, false);
  AddParameter("ProductNumber", (object)entity.ProductNumber,
               false);
  AddParameter("Color", (object)entity.Color, false);
  AddParameter("StandardCost", (object)entity.StandardCost, false);
  AddParameter("ListPrice", (object)entity.ListPrice, false);
  AddParameter("Size", (object)entity.Size ?? DBNull.Value, true);
  AddParameter("Weight", (object)entity.Weight ?? DBNull.Value,
               true);
  AddParameter("ProductCategoryID",
              (object)entity.ProductCategoryID, false);
  AddParameter("ProductModelID", (object)entity.ProductModelID,
               false);
  AddParameter("SellStartDate", (object)entity.SellStartDate,
               false);
  AddParameter("SellEndDate", (object)entity.SellEndDate
               ?? DBNull.Value, true);
  AddParameter("DiscontinuedDate", (object)entity.DiscontinuedDate
               ?? DBNull.Value, true);
  AddParameter("ModifiedDate", (object)entity.ModifiedDate, false);
}
```

Update a Product

When you are updating all the columns in the Product table create a method named Update(). This method is very similar to the Insert() method in that you call the Validate() method to verify the data prior to updating. You also create all of the parameters by calling the BuildInsertUpdateParameters() method. You add the ProductID property which is the primary key value used to update the appropriate record in the Product table.

```
public virtual int Update(Product entity)
{
  // Reset all properties
  Reset();

  // Attempt to validate the data,
  // a ValidationException is thrown if validation rules fail
  Validate<Product>(entity);

  // Create SQL to UPDATE SalesLT.Product using dynamic SQL
  SQL = "UPDATE SalesLT.Product SET Name=@Name,
           ProductNumber=@ProductNumber, Color=@Color,
           StandardCost=@StandardCost, ";
  SQL += " ListPrice=@ListPrice, Size=@Size, Weight=@Weight,
           ProductCategoryID=@ProductCategoryID,
           ProductModelID=@ProductModelID, ";
  SQL += " SellStartDate = @SellStartDate,
           SellEndDate = @SellEndDate,
           DiscontinuedDate = @DiscontinuedDate,
           ModifiedDate = @ModifiedDate ";
  SQL += " WHERE ProductID = @ProductID";

  // Create standard update parameters
  BuildInsertUpdateParameters(entity);

  // Add primary parameter to CommandObject
  AddParameter("@ProductId", (object)entity.ProductID, false);

  // Execute Query
  RowsAffected = ExecuteNonQuery();

  return RowsAffected;
}
```

Delete a Product

To delete a product from the Product table, create a Delete() method to submit a DELETE statement.

```
public virtual int Delete(Product entity)
{
  // Reset all properties
  Reset();

  // Create SQL to DELETE FROM SalesLT.Product using dynamic SQL
  SQL = "DELETE FROM SalesLT.Product WHERE ProductID = @ProductID";

  // Add primary parameter to CommandObject
  AddParameter("@ProductId", (object)entity.ProductID, false);

  // Execute Query
  RowsAffected = ExecuteNonQuery();

  return RowsAffected;
}
```

Data Validation

Validation is accomplished via data annotations and/or your own custom logic. You saw the Product entity class earlier in this article. It had several data annotations to make several properties as required. The DataManagerBase class has a Validate() method which validates those data annotations. This method can be used if you are using WPF, the Web API or Windows Forms that don't automatically check data annotations. MVC will check these annotations if you bind your properties using the Razor syntax.

In your "manager" classes, override the Validate() method. This allows you use the data annotation validation mechanism and add on your own custom validation if needed. The *ValidationMessages* property is a list of ValidationMessage objects. Each object represents either a data annotation that has failed, or one of your own custom validations has failed. If this property contains validation message objects, then a custom exception, named ValidationException, is created with this collection of validation message objects. You throw this exception which causes all the logic in either the Insert() or Update() methods to be bypassed after the call to Validate(). This ensures that no data is sent to the database if the validation fails.

```
public override bool Validate<T>(T entityToValidate)
{
  // Check all Data Annotations first
  bool ret = base.Validate(entityToValidate);

  // Cast to a Product class
  Product entity = entityToValidate as Product;

  // TODO: Add other business rules here
  if (entity.Name.Length < 2)
  {
    AddValidationMessage("Name",
      "Name must be greater than 2 characters in length.");
  }

  if (ValidationMessages.Count > 0)
  {
    throw new ValidationException(ValidationMessages);
  }

  return ret;
}
```

Since the Validate() method in the ProductManager class throws an exception, you must catch this exception somewhere. In the Insert() method in the ProductViewModel class, add a try…catch block around the call to the Insert() method of the manager class. One of the catch blocks should catch the ValidationException object. In this catch block, call the ValidationFailed() method and pass in this custom exception. The ValidationFailed() method is contained in a view model base class and does nothing more than add the validation messages from the exception object into a *ValidationMessages* property in the view model. It also sets an *IsValidationVisible* property to true, so you may show the list of validation messages on your UI.

```
public void Insert()
{
  using (ProductSPManager mgr = new ProductSPManager()) {
    try {
      RowsAffected = mgr.Insert(Entity);

      if (RowsAffected > 0) {
        ResultText = "Insert Successful" + Environment.NewLine +
             "Rows Affected: " + RowsAffected.ToString() +
              Environment.NewLine + "ProductID: " +
              Entity.ProductID.ToString() + Environment.NewLine +
              "Return_Value: " + Entity.RETURN_VALUE.ToString();
        RaisePropertyChanged("Entity");
      }
    }
    catch (ValidationException ex) {
      ValidationFailed(ex);
    }
    catch (Exception ex) {
      PublishException(ex);
    }
  }
}
```

Multiple Result Sets

If you have a stored procedure, or dynamic SQL, that returns multiple sets, you need to be able to handle those situations. Write a class that holds a collection of each result set you are going to return.

```
public class ProductAndCategory
{
  public ProductAndCategory()
  {
    Products = new List<Product>();
    Categories = new List<ProductCategory>();
  }

  public List<Product> Products { get; set; }
  public List<ProductCategory> Categories { get; set; }
}
```

Write a method in your manager class that looks like the following method.

```
public virtual ProductAndCategory MultipleResultSets()
{
  ProductAndCategory ret = new ProductAndCategory();

  // Reset all properties
  Reset();

  // Create SQL to SELECT FROM SalesLT.Product
  SQL = "SELECT * FROM SalesLT.Product;";
  SQL += "SELECT * FROM SalesLT.ProductCategory;";

  // Execute Query
  using (IDataReader dr = GetDataReader()) {
    // Use reflection to load Product data
    ret.Products = ToList<Product>(dr);
    RowsAffected = ret.Products.Count;

    dr.NextResult();

    ret.Categories = ToList<ProductCategory>(dr);
    RowsAffected += ret.Categories.Count;
  }

  return ret;
}
```

In the above code submit the SQL statement using the GetDataReader() method. Once you retrieve the results, use the ToList() method to convert the first result set in the reader into a Products list. Call the NextResult() method to move to the next result set on the reader. Once again, use the ToList() method to convert the result set into a Category list.

Transactions

If you need to execute more than one SQL statement as a unit of work, you need to use a transaction. In the following example, you are going to insert two products into the SalesLT.Product table. Both inserts must succeed for the transaction to committed to the database.

```
public void PerformTransaction()
{
  Product prod1 = new Product
  {
    Name = "Transaction 1",
    ProductNumber = "TRN-01",
    Color = "Red",
    StandardCost = 5,
    ListPrice = 10,
    Size = "Small",
    ProductCategoryID = 1,
    ProductModelID = 1,
    SellStartDate = DateTime.Now,
    ModifiedDate = DateTime.Now
  };

  Product prod2 = new Product
  {
    Name = "Transaction 2",
    ProductNumber = "TRN-02",
    Color = "Blue",
    StandardCost = 10,
    ListPrice = 20,
    Size = "Med",
    ProductCategoryID = 1, // Comment out this line to test rollback
    ProductModelID = 1,
    SellStartDate = DateTime.Now,
    ModifiedDate = DateTime.Now
  };

  // Execute Query and return DataReader
  using (ProductManager mgr = new ProductManager()) {
    using(IDbTransaction trans = mgr.BeginTransaction()) {
      try {
        // Submit the two action statements
        mgr.Insert(prod1);
        mgr.Insert(prod2);

        // Commit the transaction
        mgr.Commit();

        ResultText = "Transaction Committed";
      }
      catch (Exception ex) {
        // Rollback the transaction
        mgr.Rollback();

        ResultText = "Transaction Rolled Back";

        // Publish the exception
        PublishException(ex);
      }
    }
  }
}
```

After creating an instance of the ProductManager class call the BeginTransaction() method. The CommandObject now uses this transaction context for all statements submitted on it, until either a Commit() method or a Rollback() method is called on the transaction object.

Data Manager Base Class

Each of your "manager" classes inherits from either a SqlServerDataManagerBase class or an OracleDataManagerBase class. These classes, in turn, inherit from the DataManagerBase class. This class is an abstract class and thus cannot be instantiated. The reason is for things like parameters, commands and connections, you need concrete implementations, not just the interfaces such as IDataParameter, IDbCommand and IDbConnection.

The DataManagerBase class contains methods such as ExecuteScalar, ExecuteNonQuery, GetDataSet, and GetDataReader. Each of these methods works with the interfaces so they can be used by any .NET provider which implements these interfaces. This helps keep the code you must write in the SqlServerDataManagerBase and OracleDataManagerBase classes much smaller. If you have any other .NET provider you wish to interact with, you only need to inherit from the DataManagerBase class and override the specified methods in order to get the new provider working.

Properties

The table below lists the properties and what each one is used for.

Property Name	Type	Description
CommandObject	IDbCommand	The command object used to submit SQL to the database.
ConnectStringName	string	The name of the connection string in the <connectionStrings> element in your config file for your application. <add name="**AdventureWorksLT**" connectionString= "Server=Localhost; Database=AdventureWorksLT;..." />
ConnectionString	string	The connection string used to connect to the database.
DataSetObject	DataSet	The last DataSet filled.

IdentityGenerated	object	The value returned after an INSERT statement is submitted and the table has an auto-incrementing primary key. This value may also be filled in with an OUTPUT parameter from a stored procedure.
LastException	Exception	The last Exception thrown
LastExceptionMessage	string	The last exception message thrown
ParameterToken	string	The symbol used for parameters. For example, SQL Server uses an at sign (@). Oracle uses a colon (:).
RowsAffected	int	How many rows affected/returned from the last SQL statement submitted to the database.
SQL	string	The SQL statement to be submitted to the database.
ValidationMessages	List<ValidationMessage>	A list of validation error messages when the data in your entity object fails validation.

Methods

The following table describes the functionality of each of the methods in the DataManager base class.

Method Name	Description
AddParameter	These methods must be overridden in your provider implementation. Use these methods to add a parameter to your CommandObject as needed to support the SQL statement you are submitting.
AddStandardParameters	This abstract method needs to be overridden somewhere in your inheritance chain. It is used to add any standard parameters you might normally add to all your stored procedures.
AddValidationMessage	Prior to inserting or updating a record in a table, you should check the data in your object to ensure it is valid. If one of the properties contains errors, create a ValidationMessage object, fill in the property name and the error message to display and add that object to the *ValidationMessages* collection property.
CheckCommand	This method checks the command object to ensure it is not null, if it is, it creates a concrete implementation of a command object. It then creates a connection object and assigns the SQL property to the command object.
CountRecords	Count records from the SQL statement submitted to this method.

CreateDataAdapter	This virtual method should be overridden in your data providers class and return a concrete implementation of a data adapter object.
CreateCommand	This virtual method should be overridden in your data providers class and return a concrete implementation of a command object.
CreateConnection	This virtual method should be overridden in your data providers class and return a concrete implementation of a connection object.
CreateDbException	Creates an instance of a DataException object from all available information within properties of the DataManagerBase class.
CreateParameter	These methods must be overridden in your provider implementation. Use these methods to create a new parameter and fill in the appropriate data for that parameter.
Dispose	Closes any open connections, disposes of connections and command objects.
ExecuteScalar	If you are executing a SELECT COUNT(*) or other scalar function that returns a single value, call this method. The SQL property should be filled in with the SQL query to execute, and the Parameters collection can contain any number of Parameter objects needed to support the query you are submitting.
ExecuteNonQuery	To submit an INSERT, UPDATE or DELETE statement, either with dynamic SQL or a stored procedure use the ExecuteNonQuery() method. There are several overloads of this method available depending upon your needs. NOTE: Do NOT use the SET NOCOUNT ON statement in your stored procedure if you want to retrieve the number of rows affected by the statement in your stored procedure.
ExecuteNonQueryUsingDataSet	This method is called when you are submitting an INSERT statement using dynamic SQL and you wish to retrieve the rows affected by the statement, and the identity generated. This method adds on the following SQL statement to the existing CommandText property in the command object. SELECT @@ROWCOUNT As RowsAffected, SCOPE_IDENTITY() AS IdentityGenerated
GetParameter	If you have any OUTPUT parameters defined in your stored procedure, after you call the stored procedure, you can retrieve the parameter object using the GetParameter() method.
GetParameterValue<T>	A generic method is provided to return the value from an output parameter. Instead of having to cast the value, as just shown, you can use this method to return the value. This method must be overridden in your specific providers implementation. See the SqlServerDataManagerBase class section later in this article.
GetIdentityValue<T>	Call this method to retrieve the value of the IdentityGenerated property as a specific data type.

GetRecord<T>	This method is used to create a single object from a single row. This assumes you are submitting a SQL statement that returns a single row of data.
GetRecordUsingDataSet<T>	This method is used to create a single object from a single row. This assumes you are submitting a SQL statement that returns a single row of data. A DataSet is used to build the result set.
GetRecords<T>	Create a List<T> of rows from the SQL statement submitted to this method.
GetRecordsUsingDataSet<T>	Create a List<T> of rows from the SQL statement submitted to this method. This method builds a DataSet prior to building the List<T> of rows.
GetDataReader	For the fastest method of retrieving a result set, use a Data Reader. However, you must be careful to wrap this object into a using() block to ensure it is closed and disposed of.
GetDataSet	To retrieve a result set and have it placed into a DataSet, use the GetDataSet() method.
GetStandardOutputParameters	This abstract method needs to be overridden somewhere in your inheritance chain. It is used to retrieve any standard output parameters you might normally add to all your stored procedures.
Initialize	Called by the constructor to initialize some of the properties of this class.
Reset	Call this before submitting a new command to the database.
SetConnectionStringOrName	Pass in the name of the element in the <connectionStrings> element that is used to retrieve the connection string. Or, you can pass in a connection string to this method. Either way, the ConnectionString property is set via this call.
ToList<T>(IDataReader)	This method uses reflection to loop through all rows contained in the data reader. For each row, a new entity class is created and the field values from the data reader are set into the corresponding properties of the entity class. The new class is added to a collection of entities. You may use the [Column] attribute on the entity class if the names of your properties are different from the names of the fields in your table. If there is no column attribute, it is assumed the name of the property is the same name as the field in the table.

ToList<T>(DataTable)	This method uses reflection to loop through all rows contained in the DataTable. For each row, a new entity class is created and the field values from the DataRow are set into the corresponding properties of the entity class. The new class is added to a collection of entities. You may use the [Column] attribute on the entity class if the names of your properties are different from the names of the fields in your table. If there is no column attribute, it is assumed the name of the property is the same name as the field in the table.
ThrowDbException	If something goes wrong when submitting your SQL query to the database, you need to gather as much information about the error as possible. The ThrowDbException() method creates an instance of a DataException object using the CreateDbException() method and sets properties in this object such as the connection string, the SQL statements, the database, the data source, the parameters and the workstation id. All this information is used to generate great information for helping you debug the error. You should check out the DataException class in the sample project to see how this class formats and returns all this information. The information from this object should be used in combination with an exception publishing system so the error information generated can be stored for future debugging.
Validate	As previously mentioned, it is a good idea to use data annotations, such as [Required], [StringLength], etc., on your entity classes. If you are using ASP.NET MVC, the ModelState engine checks these attributes and will generate a collection of error objects for you if any property does not pass the validation specified in the attribute. However, other UI technologies such as WPF and Windows Forms, do not perform this automatic checking on data annotation attributes. The Validate() method in the DataManagerBase class performs the same validation as the ModelState engine in ASP.NET. It then reads the collection of validation errors and calls the AddValidationMessage() method to build the collection of ValidationMessage objects.

SqlServerDataManagerBase Class

The DataManagerBase class is abstract and thus you need to create a concrete implementation before you can use it. The SqlServerDataManagerBase class is one such implementation. This class allows you to interact with SQL Server databases.

You should open the SqlServerDataManagerBase class and look at the methods within this class to see how the concrete implementation works.

AppDataManagerBase Class

The AppDataManagerBase class should be created for each application. This class passes into the DataManagerBase class the name of the element in the <connectionStrings> element that holds the connection string. Or, you may also pass in the actual connection string.

This class is also for you to add any standard parameters you might use in every stored procedure in your application.

```
public class AppDataManagerBase : SqlServerDataManagerBase
{
  #region Constructor
  /// <summary>
  /// Pass in either a connection string, or the name in
  /// the &lt;connectionStrings&gt; element that
  /// contains the connection string.
  /// </summary>
  public AppDataManagerBase() : base("AdventureWorksLT") { }
  #endregion

  public override void AddStandardParameters()
  {
    base.AddStandardParameters();

    if (CommandObject.CommandType == CommandType.StoredProcedure) {
      // TODO: Add any standard parameters you have in your
      //       stored procedures for this application

    }
  }

  public override void GetStandardOutputParameters()
  {
    base.GetStandardOutputParameters();

    if (CommandObject.CommandType == CommandType.StoredProcedure) {
      // TODO: Add any standard OUTPUT parameters you have in your
      //       stored procedures for this application
    }
  }
}
```

Summary

In this article you learned how to create a set of wrapper classes to make it easy to call stored procedures and dynamic SQL using ADO.NET. This set of wrapper classes are very handy as they significantly cut down the amount of code you need to write. The set of classes illustrated in this article will help you to make calls to the database to retrieve data, modify data, and to work with parameters.

Sample Code

You can download the complete sample code at my website. http://www.pdsa.com/downloads. Choose "PDSA/Fairway Blog", then "ADO.NET Data Wrapper Classes" from the drop-down.