Simplify Calling Stored Procedures with the Entity Framework

Most programmers know that if you have anything other than a two table join, you should not use LINQ with the Entity Framework (EF). If you do, the resulting SQL that is submitted by EF to SQL Server tends to be inefficient. This can cause big performance problems in your application. It is better to put complicated JOIN statements into stored procedures and call the stored procedures. However, calling stored procedures using EF can be tedious. This article describes a set of wrapper classes that helps you to simplify these stored procedure calls.

Overview of EF Wrapper Classes

The sample code used to illustrate these wrapper classes are based on the AdventureWorksLT sample database that comes with SQL Server. The Product table in this database is used to perform searching, counting, selecting, inserting, updating and deleting. The following classes are in the sample application that comes with this article.

Class Name	Description
ProductViewModel	A view model class for working with product data. This class calls the ProductManager class to perform all EF operations.
AppViewModelBase	A base view model class specific for this sample.
EFViewModelBase	A view model base class located in a Common.EF.Library class library project. This class provides functionality used for any view model.
EFCommonBase	A base class located in a Common.EF.Library class library project. This class contains properties that any class that works with the Entity Framework will need. It also implements the INotifyPropertyChanged interface for WPF applications. This interface is optional.
ProductManager	This class contains all the methods used to perform searching, counting, selecting, inserting, updating, validating and deleting data contained within the Product table.

AppManager	This class is a small wrapper around the EFDataManagerBase class. It is used to create an instance of the AdventureWorksLT EF data context. This data context object is passed to the EFDataManager base class to perform all database interactions.
EFDataManagerBase	A base class located in a Common.EF.Library class library project. This class is responsible for submitting dynamic SQL or stored procedures. Parameters can (optionally) be passed regardless of the method used to query the database.

In the sample for this article, you are working with a single 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.

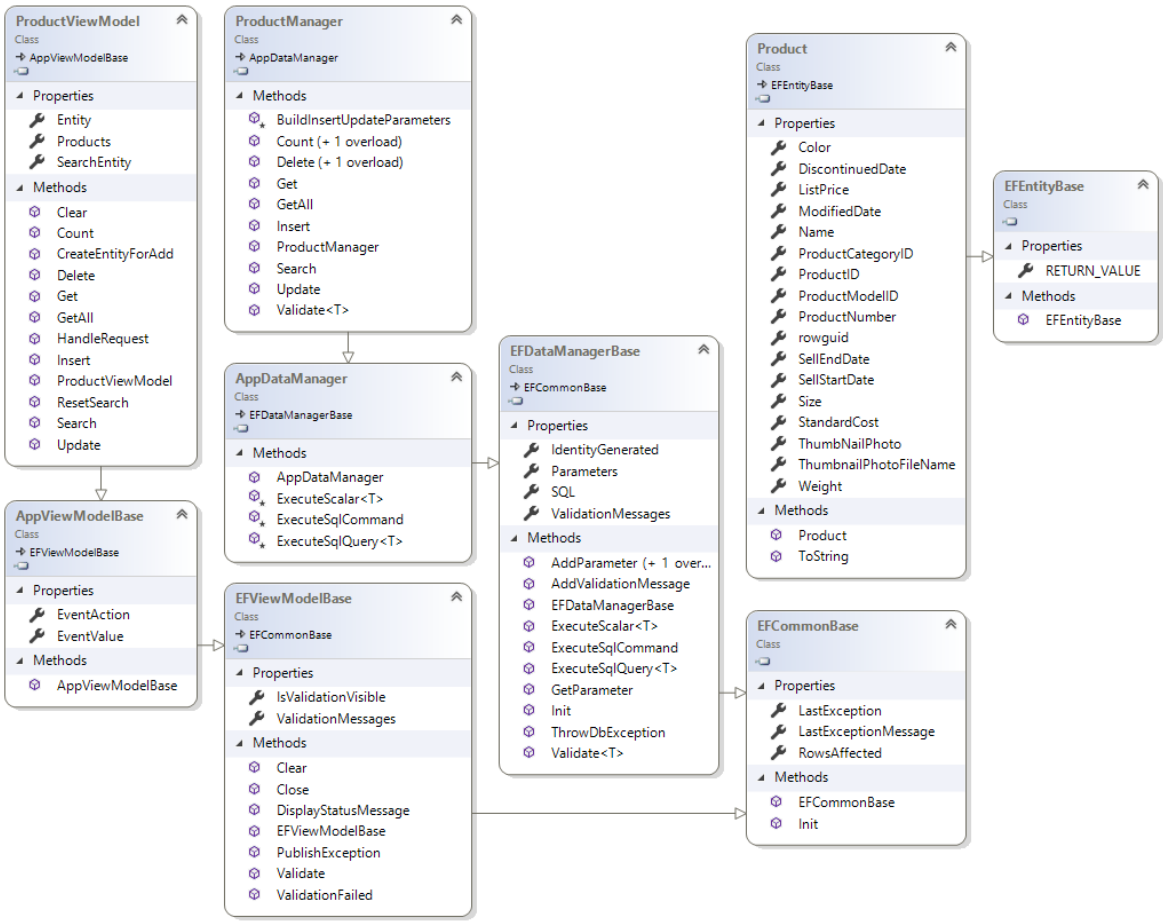

Figure 1: A diagram of the classes used to query a database using EF and stored procedures.

The App Data Manager class

Each of your "manager" classes inherits from a class named AppDataManager. This class is responsible for instantiating your EF DbContext class. In the sample for this article, this is named AdventureWorksLTDbContext as shown below.

```
public partial class AdventureWorksLTDbContext : DbContext
{
  public AdventureWorksLTDbContext() : base("name=AdventureWorksLT")
  {
  }

  public virtual DbSet<Product> Products { get; set; }

  protected override void OnModelCreating(
    DbModelBuilder modelBuilder)
  {
    // Don't let EF create migrations
    // or check database for model consistency
    Database.SetInitializer<AdventureWorksLTDbContext>(null);

    base.OnModelCreating(modelBuilder);
  }
}
```

The AppDataManager class inherits from the EFDataManagerBase class which contains methods such as ExecuteSqlCommand, ExecuteSqlQuery, and ExecuteScalar. AppDataManager overrides these methods to create an instance of the AdventureWorksLTDbContext object and pass this instance to the methods in the base class. Below is an example of just one of the methods.

```
protected List<T> ExecuteSqlQuery<T>(string exceptionMsg = "")
{
  List<T> ret = new List<T>();

  using (AdventureWorksLTDbContext db =
         new AdventureWorksLTDbContext()) {
    ret = base.ExecuteSqlQuery<T>(db, exceptionMsg);
  }

  return ret;
}
```

If you have more than one DbContext class, add additional methods in this class to create instances of those DbContext classes and submit those to the methods in the EFDataManagerBase class.

The Product Entity Class

Create an entity class and add the appropriate data annotations for the Entity Framework to use to match up the columns in the table with properties in your entity class. 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 : EFEntityBase
{
  [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() + ")";
  }
}
```

Product Manager Class

The following methods are in a class called ProductManager. In this class is where you write the code necessary to perform all actions against your Product table. For as many tables as you have in your database, you will create this many corresponding "manager" classes to retrieve data from and modify data within each table. Either create additional "manager" classes to call stored procedures that perform joins between tables or place them in existing table "manager" classes.

Get All Records

In the AdventureWorksLT database, I added a stored procedure to retrieve all products from the SalesLT.Product table. This stored procedure does not have any parameters. This stored procedure, named SalesLT.Product_GetAll looks like the following:

```
CREATE PROCEDURE SalesLT.Product_GetAll
AS
BEGIN
  SELECT *
  FROM SalesLT.Product;
END
```

To call this stored procedure, a method named GetAll() is created in the ProductManager class.

```
public List<Product> GetAll()
{
  List<Product> ret = new List<Product>();

  // Initialize all properties
  Init();

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

  // Execute Query
  ret = ExecuteSqlQuery<Product>(
   "Exception in ProductManager.GetAll()");

  return ret;
}
```

As you can see, the code to execute a stored procedure without any parameters is very simple.

Get A Single Product

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 ProductManager class.

```
public Product Get(int productId)
{
  Product ret = new Product();

  // Initialize all properties
  Init();

  // Create SQL to call stored procedure
  SQL = "SalesLT.Product_Get @ProductID";

  // Create parameters
  AddParameter("ProductID", (object)productId, false);

  // Execute Query
  var list = ExecuteSqlQuery<Product>(
    "Exception in ProductManager.Get()");
  if (list != null && list.Count > 0) {
    ret = list[0];
  }

  return ret;
}
```

When you have a stored procedure with parameters, you simply make a call to the AddParameter() method to add each parameter you need to pass into stored procedure. Each parameter must also be passed in the *SQL* property.

Search for Products Using a Stored Procedure

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 cost to search for. This data is passed to a stored procedure named Product_Search. Each of these parameters can be null if the user does not fill them in. If all the parameters are passed as null, the resulting query is the same as performing a SELECT * FROM SalesLT.Product.

```
CREATE PROCEDURE [SalesLT].[Product_Search]
 @Name nvarchar(50) null,
 @ProductNumber nvarchar(25) null,
 @BeginningCost money null,
 @EndingCost money null
AS
BEGIN
 SELECT *
 FROM SalesLT.Product
  WHERE (Name LIKE COALESCE(@Name, '') + '%')
  AND   (ProductNumber LIKE COALESCE(@ProductNumber, '') + '%')
  AND   (ISNULL(@BeginningCost, -1) = -1
         OR StandardCost >= @BeginningCost)
  AND   (ISNULL(@EndingCost, -1) = -1
         OR StandardCost <= @EndingCost)
END
```

ProductSearch Class

To hold the 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 Product_Search stored procedure, create a method named Search() in your ProductManager class as shown below.

```
public List<Product> Search(ProductSearch search)
{
  List<Product> ret = new List<Product>();

  // Initialize all properties
  Init();

  // Create SQL to call stored procedure
  SQL = "SalesLT.Product_Search @Name, @ProductNumber,
                                @BeginningCost, @EndingCost";

  // Create parameters
  AddParameter("Name", (object)search.Name ?? DBNull.Value, true);
  AddParameter("ProductNumber",
    (object)search.ProductNumber ?? DBNull.Value, true);
  AddParameter("BeginningCost",
    (object)search.BeginningCost ?? DBNull.Value, true);
  AddParameter("EndingCost",
    (object)search.EndingCost ?? DBNull.Value, true);

  // Execute Query
  ret = ExecuteSqlQuery<Product>(
   "Exception in ProductManager.Search()");

  return ret;
}
```

For each parameter in the stored procedure, pass each parameter name in the *SQL* property. In the AddParameter() 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 AddParameter() specifies if the parameter is allowed to be a null value.

Count/Search Products Using a Stored Procedure

Some other functionality you might want is the ability to count all rows within a table. Create a Product_Count stored procedure like the one shown below. This stored procedure is the same as the Product_Search stored procedure, where all the parameters can be null. If all the parameters are passed as null, the resulting query is the same as performing a SELECT COUNT(*) FROM SalesLT.Product.

```
CREATE PROCEDURE [SalesLT].[Product_Count]
 @Name nvarchar(50) null,
 @ProductNumber nvarchar(25) null,
 @BeginningCost money null,
 @EndingCost money null
AS
BEGIN
 SELECT Count(*)
 FROM SalesLT.Product
  WHERE (Name LIKE COALESCE(@Name, '') + '%')
  AND   (ProductNumber LIKE COALESCE(@ProductNumber, '') + '%')
  AND   (ISNULL(@BeginningCost, -1) = -1
         OR StandardCost >= @BeginningCost)
  AND   (ISNULL(@EndingCost, -1) = -1
         OR StandardCost <= @EndingCost)
END
```

Create a Count() method to which you pass an instance of the ProductSearch class. Use the values in this class to pass to each parameter of the Product_Count stored procedure.

```
public int Count(ProductSearch search)
{
  int ret = 0;

  // Initialize all properties
  Init();

  // Create SQL to count records
  SQL = "SalesLT.Product_Count @Name, @ProductNumber,
                               @BeginningCost, @EndingCost";

  // Create parameters
  AddParameter("Name", (object)search.Name ?? DBNull.Value, true);
  AddParameter("ProductNumber",
    (object)search.ProductNumber ?? DBNull.Value, true);
  AddParameter("BeginningCost",
    (object)search.BeginningCost ?? DBNull.Value, true);
  AddParameter("EndingCost",
    (object)search.EndingCost ?? DBNull.Value, true);

  // Execute Query
  ret = ExecuteScalar<int>(
    "Exception in ProductManager.Count(search)");

  return ret;
}
```

Count All Products Using a Stored Procedure

If you wish to get a total count of all products in the Product table, create a Count() method that creates an empty ProductSearch object and passes it to the Count() method illustrated above.

```
public int Count()
{
  return Count(new ProductSearch());
}
```

Insert a Product using a Stored Procedure

Create a stored procedure named Product_Insert to accept all fields you wish to insert into the Product table. Include an OUTPUT parameter for the primary key if the PK is an IDENTITY field and you wish to return the value generated by SQL Server.

```
CREATE PROCEDURE [SalesLT].[Product_Insert]
 @Name nvarchar(50),
 @ProductNumber nvarchar(25),
 @Color nvarchar(15) null,
 @StandardCost money,
 @ListPrice money,
 @Size nvarchar(5) null,
 @Weight decimal(8,2) null,
 @ProductCategoryID int null,
 @ProductModelID int null,
 @SellStartDate datetime,
 @SellEndDate datetime null,
 @DiscontinuedDate datetime null,
 @ModifiedDate datetime,
 @ProductID int OUTPUT
AS
BEGIN
 INSERT INTO [SalesLT].[Product]
             ([Name],[ProductNumber],[Color],[StandardCost]
             ,[ListPrice],[Size],[Weight],[ProductCategoryID]
             ,[ProductModelID],[SellStartDate],[SellEndDate]
             ,[DiscontinuedDate],[ModifiedDate])
 VALUES
             (@Name,@ProductNumber,@Color,@StandardCost,
              @ListPrice,@Size,@Weight,@ProductCategoryID,
              @ProductModelID,@SellStartDate,@SellEndDate,
              @DiscontinuedDate,@ModifiedDate);

 SELECT @ProductID = SCOPE_IDENTITY();
END
```

Create a method named Insert() that accepts an entity class of the type Product. The Product class has one property for each field in the Product table.

```
public int Insert(Product entity)
{
  // Initialize all properties
  Init();

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

  // Create SQL to call stored procedure
  SQL = "SalesLT.Product_Insert @Name, @ProductNumber, @Color,
                                @StandardCost, @ListPrice, @Size,
                                @Weight, @ProductCategoryID,
                                @ProductModelID, @SellStartDate,
                                @SellEndDate, @DiscontinuedDate,
                                @ModifiedDate,
                                @ProductID OUTPUT";

  // Create standard insert parameters
  BuildInsertUpdateParameters(entity);

  // Create parameter to get IDENTITY value generated
  AddParameter("@ProductID", -1, true, System.Data.DbType.Int32,
    System.Data.ParameterDirection.Output);

  // Execute Query
  RowsAffected = ExecuteSqlCommand(
    "Exception in ProductManager.Insert()", true, "@ProductID");

  // Get the ProductID generated from the IDENTITY
  entity.ProductID = (int)IdentityGenerated;

  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 of 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 AddParameter() method one more time to add the OUTPUT parameter for the ProductID field.

Finally, you call the ExecuteSqlCommand() method passing in a message, a true value to specify that you are including an OUTPUT parameter to retrieve for the IDENTITY value generated, and the last parameter is the name of the OUTPUT parameter.

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 using a Stored Procedure

When you are updating all the columns in the Product table, create a stored procedure named Product_Update as shown below.

```
CREATE PROCEDURE [SalesLT].[Product_Update]
 @Name nvarchar(50),
 @ProductNumber nvarchar(25),
 @Color nvarchar(15) null,
 @StandardCost money,
 @ListPrice money,
 @Size nvarchar(5) null,
 @Weight decimal(8,2) null,
 @ProductCategoryID int null,
 @ProductModelID int null,
 @SellStartDate datetime,
 @SellEndDate datetime null,
 @DiscontinuedDate datetime null,
 @ModifiedDate datetime,
 @ProductID int
AS
BEGIN
 UPDATE SalesLT.Product
 SET [Name]=@Name, ProductNumber=@ProductNumber, Color=@Color,
     StandardCost=@StandardCost, ListPrice=@ListPrice, Size=@Size,
     [Weight]=@Weight, ProductCategoryID=@ProductCategoryID,
     ProductModelID=@ProductModelID, SellStartDate=@SellStartDate,
     SellEndDate=@SellEndDate, DiscontinuedDate=@DiscontinuedDate,
     ModifiedDate=@ModifiedDate
 WHERE ProductID = @ProductID
END
```

Create a method named Update() in the ProductManager class to call this stored procedure. 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 int Update(Product entity)
{
  // Initialize all properties
  Init();

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

  // Create SQL to call stored procedure
  SQL = "SalesLT.Product_Update @Name, @ProductNumber, @Color,
                                @StandardCost, @ListPrice, @Size,
                                @Weight, @ProductCategoryID,
                                @ProductModelID, @SellStartDate,
                                @SellEndDate, @DiscontinuedDate,
                                @ModifiedDate, @ProductID";

  // Create standard update parameters
  BuildInsertUpdateParameters(entity);

  // Primary Key parameter
  AddParameter("@ProductId", (object)entity.ProductID, false);

  // Execute Query
  RowsAffected = ExecuteSqlCommand(
    "Exception in ProductManager.Update()");

  return RowsAffected;
}
```

Delete a Product using a Stored Procedure

To delete a product from the Product table, create a stored procedure named Product_Delete as shown below.

```
CREATE PROCEDURE [SalesLT].[Product_Delete]
 @ProductID int
AS
BEGIN
 DELETE FROM SalesLT.Product WHERE ProductID = @ProductID
END
```

Create a Delete() method in the ProductManager class. To allow multiple ways to delete a product, create two overloads of the Delete() method. One you pass in an integer value for the primary key, and the other is a Product object. If you have a composite key for a table in your database, using an entity object can be simpler than passing in multiple parameters to the Delete() method.

```
public int Delete(int productId)
{
  return Delete(new Product() { ProductID = productId });
}

public int Delete(Product entity)
{
  // Initialize all properties
  Init();

  // Create SQL to call stored procedure
  SQL = "SalesLT.Product_Delete @ProductID";

  // Create parameters
  AddParameter("@ProductId", (object)entity.ProductID, false);

  // Execute Query
  RowsAffected = ExecuteSqlCommand(
    "Exception in ProductManager.Delete()");

  return RowsAffected;
}
```

Multiple Result Sets

If you have a stored procedure that returns multiple sets, you need to be able to handle those situations. For example, if you have the following stored procedure.

```
CREATE PROCEDURE [SalesLT].[MultipleResults]
AS
BEGIN
 SELECT TOP 20 * FROM SalesLT.Product;

 SELECT TOP 20 SalesOrderID, SalesOrderDetailID, OrderQty,
        sd.ProductID, UnitPrice, UnitPriceDiscount, LineTotal,
            [Name], ProductNumber, Size, [Weight]
 FROM SalesLT.SalesOrderDetail sd
 INNER JOIN SalesLT.Product p ON (sd.ProductID = p.ProductID)
END
```

You write a method in your manager class that looks like the following MultipleResultSets() method.

```
public MultipleResults MultipleResultSets()
{
  DbDataReader rdr;
  MultipleResults ret = new MultipleResults();

  // Initialize all properties
  Init();

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

  // Execute Query and return DataReader
  using (AdventureWorksLTDbContext db =
           new AdventureWorksLTDbContext())
  {
    rdr = base.ExecuteReader(db,
       "Exception in MiscManager.MultipleResultSets()");

    // Get Products from first result set
    // NOTE: The "Products" parameter to the Translate() method
    //       must match a DbSet<T> property in the DbContext class
    ret.Products = ((IObjectContextAdapter)db).ObjectContext
      .Translate<Product>(rdr, "Products",
         MergeOption.AppendOnly).ToList();

    // Move to next result set
    rdr.NextResult();

    // Get Sales Order Details from second result set
    // NOTE: The "SalesOrderDetails" parameter to the Translate()
    // method must match a DbSet<T> property in the DbContext class
    ret.SalesOrderDetails = ((IObjectContextAdapter)db)
      .ObjectContext
        .Translate<SalesOrderDetail>(rdr, "SalesOrderDetails",
          MergeOption.AppendOnly).ToList();

    rdr.Close();
  }

  return ret;
}
```

In the above code you call the MultipleResults stored procedure using the ExecuteReader() method. Once you retrieve the results, use the Translate() 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 Translate() method to convert the result set into a SalesOrderDetails list. When you are finished, call the Close() method on the reader.

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()
{
  ProductManager mgr = new ProductManager();

  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 this line to test rollback
    ProductModelID = 1,
    SellStartDate = DateTime.Now,
    ModifiedDate = DateTime.Now
  };

  // Execute Query and return DataReader
  using (AdventureWorksLTDbContext db =
           new AdventureWorksLTDbContext())
  {
    using (DbContextTransaction trans =
             db.Database.BeginTransaction())
    {
      try
      {
        // Submit the two action statements
        mgr.Insert(prod1, db);
        mgr.Insert(prod2, db);

        // Commit the transaction
        trans.Commit();

        Message = "Transaction Committed";
      }
      catch (Exception ex)
      {
        // Rollback the transaction
        trans.Rollback();
        Message = "Transaction Rolled Back";
        // Publish the exception
```

```
            System.Diagnostics.Debug.WriteLine(ex);
        }
    }
  }
}
```

After creating an instance of the DbContext class, AdventureWorksLTDbContext, call the BeginTransaction() method on the Database property in that DbContext. This DbContext 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. The Insert() method in the ProductManager class has an overload that accepts a DbContext object. Pass the DbContext with the Transaction context to the Insert() method.

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 EFDataManagerBase 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 the ProductManager class, 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 EFValidationException, 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;

  // 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 EFValidationException(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 EFValidationException 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 show the list of validation messages.

```
public void Insert()
{
  ProductManager mgr = new ProductManager();

  try {
    RowsAffected = mgr.Insert(Entity);
  }
  catch (EFValidationException ex) {
    ValidationFailed(ex);
  }
  catch (Exception ex) {
    PublishException(ex);
  }
}
```

EF Common Base Class

The EFCommonBase class contains three properties; *RowsAffected*, *LastException* and *LastExceptionMessage*. TheEFCommonBase class is inherited by the EFDataManagerBase class and the ViewModelBase class.

```
public class EFCommonBase
{
  public EFCommonBase()
  {
    Init();
  }

  private Exception _LastException = null;

  public Exception LastException
  {
    get { return _LastException; }
    set {
      _LastException = value;
      LastExceptionMessage = (value == null ?
        string.Empty : value.Message);
    }
  }
  public string LastExceptionMessage { get; set; }
  public int RowsAffected { get; set; }

  public virtual void Init()
  {
    LastException = null;
    LastExceptionMessage = string.Empty;
    RowsAffected = 0;
  }
}
```

The Init() method initializes the properties of this class to default values. The Init() method is called by the constructor, and should be called to reset these values prior to making a new call to the database.

EF Data Manager Base Class

Each of your manager classes inherits from the EFDataManagerBase class. This class is the one that interacts with the *Database* property of the DbContext object to submit the SQL via the Entity Framework. It also contains methods to help you add SqlParameter objects to the SqlCommand object used to submit the SQL. This next section describes each of the properties and methods in the EFDataManagerBase class.

Properties and Init Method

In the listing below you see the declaration for the EFDataManagerBase class.

```
public class EFDataManagerBase : EFCommonBase
{
  public string SQL { get; set; }
  public object IdentityGenerated { get; set; }
  public List<SqlParameter> Parameters { get; set; }
  public List<EFValidationMessage> ValidationMessages { get; set; }

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

    SQL = string.Empty;
    IdentityGenerated = null;
    Parameters = new List<SqlParameter>();
    ValidationMessages = new List<EFValidationMessage>();
  }

  // MORE CODE HERE
}
```

The properties in the EFDataManagerBase class and their usage is described in the table below.

Property Name	Description
SQL	The dynamic SQL or a stored procedure to submit to the database.
IdentityGenerated	Holds the last IDENTITY generated if an INSERT statement is submitted, or if you return an OUTPUT parameter with the last IDENTITY generated from a stored procedure.
Parameters	A list of SqlParameter objects associated with the SQL to submit to the database.
ValidationMessages	A list of failed business rules. Each of the ValidationMessage objects in this list contain a *PropertyName* and *Message* property filled in with the appropriate information to display to the user.

Before you submit a new SQL statement to the database, you should call the Init() method so all the properties in this class are reset back to a valid starting state.

Add Parameter Methods

You may create an instance of a SqlParameter object yourself and add it to the *Parameters* property in this class, however, there two overloaded methods that can greatly simplify the process for 90% of your needs. You can see these two methods listed below. When you pass in the parameter name, you may include or exclude the @ symbol in front of the parameter name.

```
public virtual void AddParameter(string name, object value,
   bool isNullable)
{
  if (!name.Contains("@"))
  {
    name = "@" + name;
  }
  Parameters.Add(new SqlParameter { ParameterName = name,
      Value = value, IsNullable = isNullable });
}

public virtual void AddParameter(string name, object value,
  bool isNullable, DbType type,
    ParameterDirection direction = ParameterDirection.Input)
{
  if (!name.Contains("@"))
  {
    name = "@" + name;
  }
  Parameters.Add(new SqlParameter { ParameterName = name,
         Value = value, IsNullable = isNullable, DbType = type,
         Direction = direction });
}
```

Get Parameter Method

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.

```
public SqlParameter GetParameter(string name)
{
  if (!name.Contains("@"))
  {
    name = "@" + name;
  }

  name = name.ToLower();
  return Parameters.Find(p => p.ParameterName.ToLower() == name);
}
```

For example, if you have an OUTPUT parameter named @ProductId and you wish to retrieve the value for ProductId after the call to the ExecuteSqlCommand method, use the following code.

```
var pk = GetParameter("ProductId").Value;
```

ExecuteSqlCommand Method

To submit an INSERT, UPDATE or DELETE statement, either with dynamic SQL or within a stored procedure, the Entity Framework provides the ExecuteSqlCommand() method on the *Database* property. You must pass a valid instance of a DbContext object to this method, and it will submit the SQL query in the *SQL* property to the database. It also passes in any parameters in the *Parameters* collection. The *RowsAffected* property is filled in with the number of rows affected by the statement.

```
public virtual int ExecuteSqlCommand(DbContext db,
  string exceptionMsg = "")
{
  try
  {
    // Execute the dynamic SQL
    RowsAffected = db.Database.ExecuteSqlCommand(SQL,
                        Parameters.ToArray<object>());
  }
  catch (Exception ex)
  {
    ThrowDbException(ex, db, exceptionMsg);
  }

  return RowsAffected;
}
```

NOTE:	Do NOT use the SET NOCOUNT ON statement in your stored procedure if you are using a stored procedure for INSERT, UPDATE or DELETE and you wish to retrieve the number of rows affected.

ExecuteSqlQuery Method

To retrieve a list of records using a SELECT statement, the Entity Framework provides the SqlQuery() method on the *Database* property. This is a generic method, so you must provide the type of objects you wish to map the return set to. The ExecuteSqlQuery() method on the EFDataManagerBase class is defined as a generic method to which you pass in the class that each row returned from your

SQL query should map to. The *SQL* property should be filled in with the SQL query to execute, and the *Parameters* collection can contain any number of SqlParameter objects needed to support the query you are submitting.

```
public virtual List<T> ExecuteSqlQuery<T>(DbContext db,
   string exceptionMsg = "")
{
  List<T> ret = new List<T>();

  try
  {
    ret = db.Database.SqlQuery<T>(SQL,
            Parameters.ToArray<object>()).ToList<T>();
  }
  catch (Exception ex)
  {
    ThrowDbException(ex, db, exceptionMsg);
  }

  return ret;
}
```

ExecuteReader Method

If you need a DataReader object in order to loop through the data, or to populate multiple result sets, the ExecuteReader() method helps you perform these operations.

```
public virtual DbDataReader ExecuteReader(DbContext db,
  string exceptionMsg = "")
{
  DbCommand cmd;
  DbDataReader ret = null;

  try
  {
    cmd = db.Database.Connection.CreateCommand();
    cmd.CommandText = SQL;
    db.Database.Connection.Open();

    ret = cmd.ExecuteReader(CommandBehavior.CloseConnection);
  }
  catch (Exception ex)
  {
    ThrowDbException(ex, db, exceptionMsg);
  }

  return ret;
}
```

ExecuteScalar Method

If you are executing a SELECT COUNT(*) or other scalar function that returns a single value, the ExecuteScalar() method on the EFDataManagerBase class is the method to use. The *SQL* property should be filled in with the SQL query to execute, and the *Parameters* collection can contain any number of SqlParameter objects needed to support the query you are submitting.

```
public virtual T ExecuteScalar<T>(DbContext db,
  string exceptionMsg = "")
{
  T ret = default;

  try
  {
    ret = db.Database.SqlQuery<T>(SQL,
            Parameters.ToArray<object>()).SingleOrDefault<T>();
  }
  catch (Exception ex)
  {
    ThrowDbException(ex, db, exceptionMsg);
  }

  return ret;
}
```

AddValidationMessage Method

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 EFValidationMessage object, fill in the property name and the error message to display and add that object to the *ValidationMessages* collection property.

```
public EFValidationMessage AddValidationMessage(
  string propertyName, string message)
{
  EFValidationMessage ret = new EFValidationMessage
    { PropertyName = propertyName, Message = message };

  ValidationMessages.Add(ret);

  return ret;
}
```

Validate Method

When you use the Entity Framework and you create an entity class that maps properties to columns in a table, you generally use data annotations to help with validation. You may use attributes such as [Required], [StringLength], etc. 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 EFDataManagerBase 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.

```
public virtual bool Validate<T>(T entityToValidate)
{
  string propName = string.Empty;
  ValidationMessages.Clear();

  if (entityToValidate != null)
  {
    ValidationContext context =
      new ValidationContext(entityToValidate,
          serviceProvider: null, items: null);
    List<ValidationResult> results = new List<ValidationResult>();

    if (!Validator.TryValidateObject(entityToValidate, context,
                                     results, true))
    {
      foreach (ValidationResult item in results)
      {
        if (((string[])item.MemberNames).Length > 0)
        {
          propName = ((string[])item.MemberNames)[0];
        }
        AddValidationMessage(propName, item.ErrorMessage);
      }
    }
  }

  return (ValidationMessages.Count > 0);
}
```

ThrowDbException Method

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 an EFDbException object 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 EFDbException 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.

```
public virtual void ThrowDbException(Exception ex,
  DbContext db, string exceptionMsg)
{
  exceptionMsg = string.IsNullOrEmpty(exceptionMsg) ?
    string.Empty : exceptionMsg + " - ";

  EFDbException exc = new EFDbException(exceptionMsg + ex.Message,
    ex)
  {
    ConnectionString = db.Database.Connection.ConnectionString,
    Database = db.Database.Connection.Database,
    DataSource = db.Database.Connection.DataSource,
    SQL = SQL,
    SqlParameters = Parameters,
    WorkstationId = Environment.MachineName
  };

  // Set the last exception
  LastException = exc;

  throw exc;
}
```

Summary

In this article you learned how to create a set of wrapper classes to make it easy to call stored procedures using the Entity Framework. Instead of using complicated LINQ queries, it is much more efficient to write complicated queries in T-SQL in stored procedures, and just make calls to those stored procedures. The set of classes illustrated in this article will help you to make those calls. This same set of classes can be used to execute dynamic SQL using the Entity Framework too.

Sample Code

You can download the complete sample code at my https://github.com/PaulDSheriff/BlogPosts page.