# Simplifying ADO.NET Code in .NET 6: Part 2

In the last article (Simplifying ADO.NET Code in .NET 6: Part 1), you wrote code to simplify ADO.NET and map columns to properties in a class just like ORMs such as the Entity Framework do. You learned to use reflection to make creating a collection of entity objects from a data reader and take advantage of attributes such as [Column] and [NotMapped]. In this article, you’re going to refactor the code further to make it even more generic. In addition, you’ll learn to get data from a view, get a scalar value, handle multiple result sets, and call stored procedures.

## Refactor the Code for Reusability

In the last article (CODE Magazine, July/August 2022), you added methods to the ProductRepository class to read product data from the SalesLT.Product table in the AdventureWorksLT database. If you look at this code, all of it is completely generic and can be used for any table. As such, this code should be moved to a base class from which you can inherit. You can then have a ProductRepository, CustomerRepository, EmployeeRepository, and other classes that can all inherit from the base class yet add functionality that’s specific for each table.

### Create a Repository Base Class

Right mouse-click on the **Common** folder and create a new class named **RepositoryBase** and add the code shown **Listing 1**. Notice that the properties are the same as what you previously added to the ProductRepository class. The constructor for this class must be passed the generic DatabaseContext class. After setting the **DbContext** property, the Init() method is called to initialize all the properties to a valid start state.

### Add Search() Method Just for Products

Add a Search() method to the RepositoryBase class just below the Init() method. This method is different from the Search() method previously written in the ProductRepository class because it removes the **using** around the SqlServerDatabaseContext.

public virtual List<TEntity>

C Search<TEntity>() {

List<TEntity> ret;

// Build SQL from Entity class

SQL = BuildSelectSql<TEntity>();

// Create Command Object with SQL

DbContext.CreateCommand(SQL);

// Get the list of entity objects

ret = BuildEntityList<TEntity>

(DbContext.CreateDataReader());

return ret;

}

You now need to move the BuildEntityList(), BuildCollumnCollection() and the BuildSelectSql() methods from the ProductRepository class into this new RepositoryBase class.

### Simplify the Product Repository Class

Now that you have a RepositoryBase class with all of the methods moved from the ProductRepository class, you can greatly simplify the ProductRepository class by having it inherit from the RepositoryBase class. Modify the **ProductRepository.cs** file to look like **Listing 2**.

In the ProductRepository class you must accept a database context object in the constructor because without one, there’s no way you could interact with the Product table. A specific Search() method is created to return a list of Product objects in the ProductRepository class, but it simply uses the generic Search<TEntity>() method from the RepositoryBase class.

### Add Database Context Class for the AdventureWorksLT Database

Instead of using the generic DatabaseContext or SqlServerDatabaseContext classes directly, it’s a better practice to create a database context class for each database you wish to interact with. Right mouse-click on the project and add a new folder named **Models**. Right mouse-click on the Models folder and add a new class named **AdvWorksDbContext** that inherits from the SqlServerDatabaseContext class, as shown in **Listing 3**.

The AdvWorksDbContext class inherits from the SqlServerDatabaseContext because the AdventureWorksLT database you are interacting with is in a SQL Server. An instance of the ProductRepository class is created in the Init() method and exposed as a public property named **Products**. The AdvWorksDbContext is passed to the constructor of the ProductRepository class because it needs the services of a database context to perform its functions against the Product table.

### Try It Out

Now that you have made these changes, let's ensure that you can still retrieve all records from the Product table. Open the **Program.cs** file and add a new using statement at the top of the file.

using AdoNetWrapperSamples.Models;

Remember that you removed the **using** from the Search() method in the RepositoryBase class? You’re now going to create the **using** wrapper around the AdvWorksDbContext class to have all objects disposed of properly once you’ve retrieved all records.

Remove all the lines of code from where you create the ProductRepository class and the call to the Search() method. Add in the code shown in the snippet below. You can now see the **using** statement that wraps up the instance of the AdvWorksDbContext class. This code should look familiar if you have used the Entity Framework (EF), as this is typically how you interact with DbContext classes you create with EF.

**using AdvWorksDbContext db = new(ConnectString);**

List<Product> list = **db.Products.Search()**;

Console.WriteLine("\*\*\* Get Product Data \*\*\*");

// Display Data

foreach (var item in list) {

Console.WriteLine(item.ToString());

}

Console.WriteLine();

Console.WriteLine(

$"Total Items: {list.Count}");

Console.WriteLine();

**Console.WriteLine(**

**$"SQL Submitted: {db.Products.SQL}");**

Console.WriteLine();

Run the console application and you should see the complete list of product objects displayed. In addition, you should see the SQL statement submitted by the classes you created in this article.

## Searching for Data

In addition to retrieving all records, you probably want to add a WHERE clause to filter the records based on some condition. For example, you might wish to locate all Product records where the **Name** column starts with a specific character and the **ListPrice** column contains a value greater than a specific value. You want to have the wrapper classes generate a SQL statement that looks like the following.

SELECT \* FROM SalesLT.Product

WHERE Name LIKE @Name + '%'

AND ListPrice >= @ListPrice

You need to add some new functionality to create this SQL statement. You need to pass in values to fill into the **@Name** and **@ListPrice** parameters. You also need to specify what the operators (=, LIKE, or >=) are for each expression. For example, you need to put a LIKE operator for the **@Name** parameter and a greater-than or equal-to (>=) operator for the **@ListPrice** parameter.

### Add a Product Search Class

To pass in the values to the Search() method, create a class to hold the parameters you wish to use for the WHERE clause. Right mouse-click on the project and add a new folder named **SearchClasses**. Right mouse-click on the SearchClasses folder and add a new class named **ProductSearch** that looks like the code below.

#nullable disable

using AdoNetWrapper.Common;

namespace AdoNetWrapperSamples.SearchClasses;

public class ProductSearch {

[Search("LIKE")]

public string Name { get; set; }

[Search(">=")]

public decimal? ListPrice { get; set; }

}

Create the **Name** and **ListPrice** properties to use for searching. All properties in this class should be nullable unless you wish to require the user to enter at least one search value prior to searching for records. All properties should be decorated with the [Search] attribute unless you just wish to use an equal (=) operator in the WHERE clause.

### Add a Search Attribute Class

Microsoft doesn’t have a [Search] attribute, so it’s up to you to create one. Right mouse-click on the **Common** folder and add a new class named **SearchAttribute**, as shown in the following code snippet.

#nullable disable

namespace AdoNetWrapper.Common;

[AttributeUsage(AttributeTargets.Property)]

public class SearchAttribute : Attribute {

public string SearchOperator { get; set; }

public string ColumnName { get; set; }

public SearchAttribute(string searchOperator) {

SearchOperator = searchOperator ?? "=";

}

}

There are two properties needed for this attribute class, **SearchOperator** and **ColumnName**. The **SearchOperator** property is assigned to an equal sign (=) if one isn’t supplied. If the **ColumnName** property is a null, the code you’re going to use to create the WHERE clause will use the property name of the search class.

### Modify the ColumnWrapper Class

When building the collection of columns needed for the WHERE clause, the process is going to be like the code used to build the columns for the SELECT statement. However, you’re going to need two additional items to keep track of: the value to supply as a parameter and for the search operator to use. Open the **ColumnMapper.cs** file in the Common folder and add a **ParameterValue** property and a **SearchOperator** property.

public class ColumnMapper {

public string ColumnName { get; set; }

public PropertyInfo PropertyInfo { get; set; }

public object ParameterValue { get; set; }

public string SearchOperator { get; set; }

}

### Add Method to Build Search Column Collection

Open the **RepositoryBase.cs** file and add a new method named BuildSearchColumnCollection(), as shown in **Listing 4**. This method is just like the BuildColumnCollection() method you wrote in the last article. Create an array of PropertyInfo objects for each property in the TSearch class. Loop through the array of properties and retrieve the value for the current property of the search class. If the value is filled in, create a new ColumnMapper object. Check for a [Search] attribute and if found, see if the **ColumnName** and/or the **SearchOperator** property exists. Override those properties in the ColumnWrapper object if they do exist. Add the new ColumnWrapper object into the **ret** variable to be returned once all properties in the search class are processed.

### Add Method to Create WHERE Clause for Searching

The next new method is used to build the actual WHERE clause to be added to the SELECT statement. Add a new method named **BuildSearchWhereClause()**, as shown in **Listing 5**. Pass to this method the list of ColumnWrapper objects created using the BuildSearchColumnCollection() method. Iterate over the list of objects and build the WHERE clause. Be careful when copying the code from the article as I had to break lines in the sb.Append() due to formatting of the article. The interpolated string belongs all on one line with a space between each item except between the **ParameterPrefix** and the **ColumnName** properties.

### Add Method to Create Parameters for Command Object

The last new method to build is called BuildWhereClauseParameters(), as shown in **Listing 6**. In this method, you iterate over the same collection of ColumnMapper objects you created in the BuildSearchColumnCollection() method. Each time through, build a new SqlParameter object passing in the column name and either the value to submit by itself, or if the **SearchOperator** property is equal to "LIKE", you use the value and add on a percent sign (%).

### Overload Search() Method to Accept a Command Object

Add a new overload for the Search() method to accept a Command object (**Listing 7**). This Search() method checks to ensure that the Columns collection has been built from the TEntity class. It then sets the **DbContext*.*CommandObject** property to the **cmd** object variable passed in. The BuildEntityList() method is then called to create the list of entity objects.

Modify the original Search() method to call the new overload you just created, as shown in the following code snippet.

public virtual List<TEntity> Search<TEntity>() {

// Build SQL from Entity class

SQL = BuildSelectSql<TEntity>();

// Create Command Object with SQL

DbContext.CreateCommand(SQL);

**return Search<TEntity>(DbContext.CommandObject);**

}

### Overload Search() Method to Accept Search Class

Open the **RepositoryBase.cs** file and add another overloaded Search() method that takes two type parameters TEntity and TSearch, as shown in **Listing 8**. After building the SELECT statement, call the BuildSearchColumnCollection() method that uses the TSearch class to build a collection of columns to be used in the WHERE clause. If there are any search columns, call the BuildSearchWhereClause() to build the actual WHERE clause to add to the SELECT statement. The SqlCommand object is built using the new SELECT clause, and then parameters are added with the values from the TSearch object. The SqlCommand object is then passed to the Search() method that accepts the command object.

### Modify Product Repository Class

Now that you have the generic version of the Search() method to accept a search entity object, you need to add a Search() method to the ProductRespository class to accept a ProductSearch class. Open the **ProductRepository.cs** file and add a new using statement at the top of the file.

using AdoNetWrapperSamples.SearchClasses;

Add a new Search() method to the ProductRepository class to call the Search<TEntity, TSearch>() method in the RepositoryBase class.

public virtual List<Product>

Search(ProductSearch search) {

return base

.Search<Product, ProductSearch>(search);

}

### Try It Out

Open the **Program.cs** file and add a new **using** statement at the top of the file so you can use the ProductSearch class.

using AdoNetWrapperSamples.SearchClasses;

Add an instance of the ProductSearch class and initialize the **Name** property to the value "C", and the **ListPrice** property to be 50. Call the overloaded Search() method you just added to the ProductRepository class and pass in the instance of the ProductSearch class as shown in the following code.

using (AdvWorksDbContext db = new(ConnectString));

**ProductSearch search = new() {**

**Name = "C",**

**ListPrice = 50**

**};**

List<Product> list =

db.Products.Search(**search**);

// REST OF THE CODE HERE

Run the console application and you should see three products displayed, as shown in **Figure 1**.

Text

Description automatically generated

**Figure 1:** Build a WHERE clause to limit the total records returned

## Create Generic Method to Submit SQL

Sometimes you may need a way to submit any SQL statement to the database and have it return any list of objects you want. Maybe you want to submit some SQL that has a few tables joined together. Into which repository class would you want to put that? Instead of worrying about where it belongs, you can create a **Database** property on the AdvWorksDbContext class that’s of the type RepositoryBase and just submit the SQL using a SqlCommand object. Open the **AdvWorksDbContext.cs** file and add a new property of the type RepositoryBase.

public RepositoryBase Database { get; set; }

Modify the constructor of the AdvWorksDbContext class to pass in the current instance of AdvWorksDbContext to the RepositoryBase class instance called **Database**.

public virtual void Init() {

**Database = new(this);**

Products = new(this);

}

### Building Your Own Command Object

Open the **Program.cs** file and create a SQL string with the same WHERE clause you created earlier (**Listing 9**). Create a SqlCommand object by calling the CreateCommand() method and pass in the **sql** variable. Add the parameters to the command object and pass in some hard-coded values. Call the Search<Product>(cmd) method directly to retrieve the list of rows in the Product table that match the search criteria.

### Try It Out

Run the console application and you should see three products displayed, as shown in **Figure 2**.

A screenshot of a computer

Description automatically generated

**Figure 2:** Add a WHERE clause to your SQL by using a search class and the [Search] attribute.

## Retrieve Data from a View

Now let's retrieve the data from a view in the AdventureWorksLT database named vProductAndDescription. If this view isn’t already in the AdventureWorksLT database, create it using the following SQL:

CREATE VIEW vProductAndDescription AS

SELECT p.ProductID, p.Name,

pm.Name AS ProductModel,

pmd.Culture, pd.Description

FROM SalesLT.Product AS p

INNER JOIN SalesLT.ProductModel AS pm

ON p.ProductModelID = pm.ProductModelID

INNER JOIN

SalesLT.ProductModelProductDescription AS pmd

ON pm.ProductModelID = pmd.ProductModelID

INNER JOIN

SalesLT.ProductDescription AS pd

ON pmd.ProductDescriptionID =

pd.ProductDescriptionID;

Add a new class named ProductAndDescription to map to the vProductAndDescription view. Right mouse-click on the EntityClasses folder and add a new class named **ProductAndDescription**, as shown in **Listing 10**.

### Try It Out

Open the **Program.cs** file and modify the code to call the view using the Search() method on the Database property.

using AdvWorksDbContext db = new(ConnectString);

// Get all rows from view

List<ProductAndDescription> list =

db.Database.Search<ProductAndDescription>();

Console.WriteLine("\*\*\* Get Product Data \*\*\*");

// Display Data

foreach (var item in list) {

Console.WriteLine(item.ToString());

}

Console.WriteLine();

Console.WriteLine(

$"Total Items: {list.Count}");

Console.WriteLine();

Console.WriteLine(

$"SQL Submitted: {db.Database.SQL}");

Run the console application and you should see over 1700 rows appear from the view. Many of these have a bunch of questions marks. This is because the data in the table has some foreign language characters.

## Search Using a View

Just like you created a search class for the Product table, you can also create a search class for searching when using a view. Right mouse-click on the SearchClasses folder and add a new class named **ProductAndDescriptionSearch**, as shown in the code snippet below.

#nullable disable

using AdoNetWrapper.Common;

namespace AdoNetWrapperSamples.SearchClasses;

public class ProductAndDescriptionSearch {

[Search("=")]

public string Culture { get; set; }

}

### Try It Out

Modify the code in **Program.c**s to create an instance of this new search class. Set the **Culture** property to the value "en" so you only grab those records where the Culture field matches this value. Call the overload of the Search() method to which you pass a search class.

ProductAndDescriptionSearch search = new() {

Culture = "en",

};

// Perform a search for specific culture

List<ProductAndDescription> list =

db.Database.Search<ProductAndDescription,

ProductAndDescriptionSearch>(search);

Run the console application and you should see almost 300 rows of data returned from the view.

## Find a Single Product

Now that you’ve learned how to create a WHERE clause, you can use this same kind of code to locate a record by its primary key. The ProductID column in the SalesLT.Product table is the primary key, so you want to create a SELECT statement that looks like the following:

SELECT \* FROM SalesLT.Product

WHERE ProductID = @ProductID

### Use the [Key] Attribute

To do this, you must identity the property in the Product class that holds the primary key. You’re going to do this using the [Key] attribute class that .NET provides. Open the **Product.cs** file and add a using statement.

using System.ComponentModel.DataAnnotations;

Add the [Key] attribute above the **Id** property.

[Key]

[Column("ProductID")]

public int Id { get; set; }

Open the **ColumnMapper.cs** file and add a new property called ***IsKeyField*** so that as you are looping through and building the list of properties, you can set this Boolean property to true for the property decorated with the [Key] attribute.

public bool IsKeyField { get; set; }

Open the **RepositoryBase.cs** file and add a using statement at the top of the file.

using System.ComponentModel.DataAnnotations;

Locate the BuildColumnCollection() method and just below the code where you check for a ColumnAttribute and set the colMap.ColumnName, add the following code to check for the [Key] attribute:

// Is the column a primary [Key]?

KeyAttribute key = prop

.GetCustomAttribute<KeyAttribute>();

colMap.IsKeyField = key != null;

### Add a Find() Method

Add a new method named Find() to the RepositoryBase class, as shown in **Listing 11**. This method has the same signature as the LINQ Find() method, where you pass in one or more values to a parameter array. Most tables only have a single field as their primary key, but in case a table has a composite key, you need to have a parameter array for those additional values.

The BuildSelectSql() method creates the SELECT statement, and the **Columns** property. Next, the **searchColumns** variable is created as a list of ColumnMapper objects with just those columns where the **IsKeyField** property is set to true. Ensure that the number of values passed into the parameter array are equal to the number of properties with the [Key] attribute. If these two numbers don’t match, throw an ApplicationException object.

Loop through the collection of **searchColumns** and fill in the **ParameterValue** property for each ColumnWrapper object in the list. Set the **SearchOperator** property for each to be an equal sign because you’re looking for an exact match.

Build the WHERE clause for the SELECT statement by using the BuildSearchWhereClause() method you created earlier. Build the SqlCommand object and then build the parameters for the WHERE clause by calling the BuildWhereClauseParameters() method.

Call the overload of the Find() method shown in **Listing 12**. This method is responsible for passing the command object to the Search() method and retrieving the results back. Check the results to ensure values were found, and if there’s at least one product in the list, assign the first item to the **ret** variable to be returned from this method. If no values are found, a null value is returned just like the LINQ Find() method.

Now that you have the generic Find() methods written in the RepositoryBase class, open the **ProductRepository.cs** file and add the Find() method that accepts an integer value that relates to the ProductID field in the Product table.

public virtual Product Find(int id) {

return base.Find<Product>(id);

}

### Try It Out

Open the **Program.cs** file and change the code to call the Find() method, as shown in **Listing 13**. This method should check to ensure that a single entity class is returned. If the value returned is null, write a message into the console window, otherwise, write the product entity into the console window. Run the console application and you should see a single product object displayed. You may need to change the product ID to match an ID from your SalesLT.Product table.

## Get a Scalar Value

If you need to retrieve the value from one of the many aggregate functions in SQL Server, such as Count(), Sum(), Avg(), etc., expose a method named ExecuteScalar() from the RepositoryBase class. To retrieve the count of all records in the Product table, submit a SQL statement such as the following:

SELECT COUNT(\*) FROM SalesLT.Product;

Place this SQL statement into a Command object and call the ExecuteScalar() method on the Command object. Open the **RepositoryBase.cs** file and add a new method. Because you don’t know what type of object you’re going to get back, return an object data type.

public virtual object

ExecuteScalar(IDbCommand cmd) {

object ret;

// Open the Connection

DbContext.CommandObject.Connection.Open();

// Call the ExecuteScalar() method

ret = DbContext.CommandObject.ExecuteScalar();

return ret;

}

Add an overload of the ExecuteScalar() method to allow you pass in a simple SQL statement. This method then creates the Command object and passes it to the previous ExecuteScalar() overload for processing.

public virtual object

ExecuteScalar(string sql) {

// Store the SQL submitted

SQL = sql;

// Create Command object with SQL

DbContext.CreateCommand(SQL);

// Return the value

return ExecuteScalar(DbContext.CommandObject);

}

### Try It Out

Open the **Program.cs** file and add code to test this out.

using AdvWorksDbContext db = new(ConnectString);

string sql = "SELECT COUNT(\*)

FROM SalesLT.Product";

int rows = (int)db.Database.ExecuteScalar(sql);

Console.WriteLine(

"\*\*\* ExecuteScalar(sql) Sample \*\*\*");

// Display Result

Console.WriteLine(rows);

Console.WriteLine();

Console.WriteLine(

$"SQL Submitted: {db.Database.SQL}");

Console.WriteLine();

Run this application and you should see the total number of products within the Product table appear in the console window.

## Multiple Result Sets

Sometimes, retrieving multiple result sets can help you cut down the number of roundtrips to your SQL Server. A data reader object supports reading one result set and then advancing to the next. Let's look at how this works with the wrapper classes you’ve created so far.

### Create New Search() Method Overload

Open up the **RepositoryBase.cs** file and create a new overload of the Search() method, as shown in **Listing 14**. This method accepts both a command object and a data reader, and it’s responsible for calling the BuildEntityList() method.

Modify the old Search() method to have it now call this new overload, as shown in the code snippet below. Remove the declaration of the **ret** variable, and modify the return statement to call the new overloaded Search() method.

public virtual List<TEntity> Search<TEntity>

(IDbCommand cmd) {

// Build Columns if needed

if (Columns.Count == 0) {

Columns = BuildColumnCollection<TEntity>();

}

// Set Command Object

DbContext.CommandObject = cmd;

**return Search<TEntity>(cmd,**

**DbContext.CreateDataReader());**

}

### Add a Customer Entity Class

To illustrate multiple result sets, you need a new entity class. In the AdventureWorksLT database, there’s a Customer table. Let's create a new **Customer.cs** file and add the code shown in **Listing 15** to model that table.

### Add a View Model Class

Instead of writing the code to handle multiple result sets in the Program.cs file, create a new view model class to encapsulate the functionality of reading both product and customer data. Right mouse-click on the project and create a folder named **ViewModelClasses**. Right mouse-click on the ViewModelClasses folder and add a new class named **ProductCustomerViewModel.cs** and add the code shown in **Listing 16**.

The code in the LoadProductsAndCustomers() method creates a string with two SQL statements in it. An instance of the AdvWorksDbContext class is created with a **using** block so all connection objects are disposed of properly. Next a SqlCommand object is created by calling the CreateCommand() method on the database context object.

The Search<Product>() method is called to load the set of product data. Call the NextResult() method on the data reader object to move to the next result set. Clear the current list of ColumnWrapper objects because that list of columns is for the Product data set. Finally, call the Search<Customer>() method passing in the command object and the current data reader object, which is now ready to loop through the customer records.

### Try It Out

To try this code out to make sure it works, open the **Program.cs** file. Put the code shown below just after the code that retrieves the connection string.

ProductCustomerViewModel vm = new(ConnectString);

vm.LoadProductsAndCustomers();

// Display Products

foreach (var item in vm.Products) {

Console.WriteLine(item);

}

// Display Customers

foreach (var item in vm.Customers) {

Console.WriteLine(item);

}

Run the application and you should see the list of products and customers appear in the console window.

## Search for Data Using a Stored Procedure

Another common method of retrieving data from a database is to call a stored procedure. If you have a three (or more) table join, it’s a best practice to move that code to a stored procedure or a view in your database. Keeping complicated queries out of your C# code is better for readability and maintenance. It also allows you to tune the join in the server. Let's look at calling a stored procedure using the ADO.NET wrapper classes. Create a stored procedure in the AdventureWorksLT database named **Product\_Search**, as shown in **Listing 17**.

### Create Parameter Class for Calling a Stored Procedure

Because the Product\_Search stored procedure has four parameters, you should create a class with four properties. Right mouse-click on the project and add a new folder named **ParameterClasses**. Right mouse-click on the ParametersClasses folder and add a new class named **ProductSearchParam**. The property names should match the parameter names within the stored procedure.

#nullable disable

using AdoNetWrapper.Common;

namespace AdoNetWrapperSamples.ParameterClasses;

public class ProductSearchParam {

public string Name { get; set; }

public string ProductNumber { get; set; }

public decimal? BeginningCost { get; set; }

public decimal? EndingCost { get; set; }

}

### Add Method to Call Stored Procedure

Open the **RepositoryBase.cs** file and create a new method named SearchUsingStoredProcedure(), as shown in **Listing 18**. In this method, pass in an instance of the parameter class and a SQL string that contains the name of the stored procedure. Assign the SQL string passed to the **SQL** property and build the columns collection for the entity class collection to be returned.

Create the command object and assign the **CommandType** property of the command object to the enumeration **CommandType.StoredProcedure**. Check the **param** parameter to ensure that it isn’t null. If not, build the collection of search columns to use to build the set of parameters that will be passed to the stored procedure. You can use the same BuildWhereClauseParameters() method you used before, as this adds parameters to the command object based on the set of ColumnWrapper objects passed to it. Finally, call the stored procedure and use the result set to build the collection of entity objects.

When you were building the WHERE clause for a dynamic SQL statement, you only needed to create ColumnWrapper object for those properties in the search class that had a value in them. When calling a stored procedure, you need to create a ColumnWrapper object for all parameters whether or not there is a value in them. Locate the BuildSearchColumnCollection() method and within the foreach() loop, modify the **if** statement that checks to see if the **value is not null** to look like the following.

if (value != null **||**

**(DbContext.CommandObject != null &&**

**DbContext.CommandObject.CommandType ==**

**CommandType.StoredProcedure)**) {

One more location you need to change code to support calling stored procedures is within the BuildWhereClauseParameters() method. As you loop through each ColumnWrapper object to build the parameter, you’re going either set the parameters' **Value** property to the value from the search class, or a **DBNull.Value**. Also change it so the **ParameterValue** property is set back into the collection of entity columns only if you are not calling a stored procedure. This is because the parameter names passed to the stored procedure may not be the same names as the property names in the entity column collection. Modify the BuildWhereClauseParameters() method to look like the code shown in **Listing 19**.

### Try It Out

Open the **Program.cs** file and modify the code after retrieving the connection string to look like **Listing 20.** Run the console application and you should see only products with names starting with the letter C appearing in the console window.

### Call Stored Procedure with No Parameters

If you have a stored procedure that doesn’t have any parameters, you can call that as well. Just pass a null value as the first parameter to the new Search() overload you just added. As an example, create the following stored procedure in the AdventureWorksLT database:

CREATE PROCEDURE [SalesLT].[Product\_GetAll]

AS

BEGIN

SELECT \*

FROM SalesLT.Product;

END

### Try It Out

Open the **Program.cs** file and modify the line of code that sets the name of the stored procedure to call.

string sql = "SalesLT.Product\_GetAll";

Next, modify the line of code that calls the SearchUsingStoredProcedure() method. The TEntity and TParam types passed should both be the Product entity class. Pass a null value to the first parameter to avoid creating any parameters for this stored procedure call.

List<Product> list = db.Database

.SearchUsingStoredProcedure

<Product**, Product**>(**null,** sql);

Run the console application and you should see all of the product data displayed after making this call to the stored procedure.

## Stored Procedure with Output Parameter

Stored procedures can not only have input parameters, but output parameters as well. To retrieve the value from an OUTPUT parameter, you need to ensure that you read the parameter immediately after calling the stored procedure. If you’re reading data using a data reader, you need to close the reader, but NOT close the connection. To test this, create the following stored procedure in the AdventureWorksLT database:

CREATE PROCEDURE

[SalesLT].[Product\_GetAllWithOutput]

@Result nvarchar(10) OUTPUT

AS

BEGIN

SELECT \*

FROM SalesLT.Product;

/\* Set the output parameter \*/

SELECT @Result = 'Success';

END

### Create [OutputParam] Attribute

You need to inform the RepositoryBase class if you’re going to have an OUTPUT parameter that needs to be returned. An easy way to do this is to create another attribute. Right mouse-click on the **Common** folder, create a new class named **OutputParamAttribute**, and enter the code shown below in this new file.

#nullable disable

using System.Data;

namespace AdoNetWrapper.Common;

[AttributeUsage(AttributeTargets.Property)]

public class OutputParamAttribute:Attribute {

public ParameterDirection Direction

{ get; set; }

public DbType DbType { get; set; }

public int Size { get; set; }

public OutputParamAttribute(

ParameterDirection direction) {

Direction = direction;

}

}

The OutputParamAttribute class inherits from the Attribute class and exposes three public properties. The **Direction** property is the one exposed from the constructor, as that’s the one you’re going to use the most.

### Create Search Class with OutputParam Attribute

Any time you have a stored procedure with parameters, you need to build a parameter class to map to those parameters. Right mouse-click on the **ParameterClasses** folder, create a new class named **ProductGetAllParam**, and enter the code shown below into this new file. Notice that the **Result** property is decorated with the new [OutputParam] attribute you just created.

#nullable disable

using AdoNetWrapper.Common;

using System.Data;

namespace AdoNetWrapperSamples.ParameterClasses;

public class ProductGetAllParam {

[OutputParam(ParameterDirection.Output,

Size = 10)]

public string Result { get; set; }

}

### Modify ColumnMapper Class

Because you now have additional properties within the [OutputParam] attribute, you need to add these same properties to the ColumnMapper class. As you iterate over the properties for a search class, you can store the data from the [OutputParam] attribute into the ColumnMapper object for use when calling the stored procedure. Open the **ColumnMapper.cs** file and add a Using statement.

using System.Data;

Add the following new properties to the ColumnWrapper class.

public ParameterDirection Direction

{ get; set; }

public DbType DbType { get; set; }

public int Size { get; set; }

Add a constructor to the ColumnMapper class to set the default parameter direction to **Input**. Also take this opportunity to initialize the ***SearchOperator*** the equal sign (=).

public ColumnMapper() {

SearchOperator = "=";

Direction = ParameterDirection.Input;

}

### Modify the BuildSearchColumnCollection() Method

Open the **RepositoryBase.cs** file and modify the BuildSearchColumnCollection() method to check for an [OutputParam] attribute. If one is found, transfer the properties found in the OutputParam into the ColumnWrapper object. Within the foreach loop, after the code that checks for a [Search] attribute, add the following code to check for an [OutputParam] attribute.

// Does Property have an [OutputParam] attribute

OutputParamAttribute oa = Prop

.GetCustomAttribute<OutputParamAttribute>();

if (oa != null) {

colMap.Direction = oa.Direction;

colMap.DbType = oa.DbType;

colMap.Size = oa.Size;

}

### Modify the BuildSearchWhereClause() Method

Now locate the BuildSearchWhereClause() method and modify the code in the foreach() to only retrieve those columns where the **Direction** property is either **Input** or **InputOutput**. Those properties that have a **Direction** set to **Output** don’t need to be included in the WHERE clause.

foreach (var item in columns

**.Where(c => c.Direction ==**

**ParameterDirection.Input**

**|| c.Direction ==**

**ParameterDirection.InputOutput)**) {

### Modify the BuildWhereClauseParameters() Method

Find the BuildWhereClauseParameters() method and modify the foreach() to only retrieve those columns where the **Direction** property is either **Input** or **InputOutput**.

foreach (ColumnMapper item in whereColumns

**.Where(c => c.Direction ==**

**ParameterDirection.Input**

**|| c.Direction ==**

**ParameterDirection.InputOutput)**) {

### Add a BuildOutputParameters() Method

For working with stored procedure OUTPUT parameters, build a new method to handle those columns in the search class that are decorated with the [OutputParam] attribute. Create a new method named BuildOutputParameters that accepts a Command object and a list of columns from the search class. In the foreach() iterator, you’re only going to extract those columns where the **Direction** property is either **Output** or **InputOutput**.

protected virtual void BuildOutputParameters

(IDbCommand cmd, List<ColumnMapper> columns) {

// Add output parameters

foreach (ColumnMapper item in columns

.Where(c => c.Direction ==

ParameterDirection.Output ||

c.Direction ==

ParameterDirection.InputOutput)) {

var param = DbContext.CreateParameter(

item.ColumnName, null);

param.Direction = item.Direction;

param.DbType = item.DbType;

cmd.Parameters.Add(param);

}

}

### Add GetOutputParameters() Method

After the stored procedure has been processed is when you may retrieve any OUTPUT parameters. Create a new method named GetOutputParameters() (shown in **Listing 21**) to iterate over the search columns and retrieve the value from the stored procedure and place it into the appropriate property of the search class.

### Create SqlServerRespositoryBase Class

When using SQL Server to retrieve OUTPUT parameters, you must set the **Size** property when adding the parameter to the Command object. This might not be true for all .NET data providers, but you need it for SQL Server. Unfortunately, the **Size** parameter does not exist on the IDbCommand interface, so you must create a SqlServerRepositoryBase class that inherits from the RepositoryBase class and override the BuildOutputParameters() method. Within this override, you set the **Size** property on the parameter object. Right mouse-click on the **Common** folder and add a new class named **SqlServerRepositoryBase**. Place the code shown in **Listing 22** into this new file.

### Modify SearchUsingStoredProcedure() Method

Open the **RepositoryBase.cs** file and locate the SearchUsingStoredProcedure() method. Within the If statement (**Listing 23**) that checks that the ***param*** variable is not null, add a new If statement immediately after the existing If statement.

Move a little further down in this method and, just after the call to the BuildEntityList() method and before the **return** statement, add the following code to retrieve any output parameters:

// Retrieve Any Output Parameters

if (searchColumns.Where(c => c.Direction ==

ParameterDirection.Output ||

c.Direction ==

ParameterDirection.InputOutput).Any()) {

// Must close DataReader for output

// parameters to be available

DbContext.DataReaderObject.Close();

GetOutputParameters(param, searchColumns);

}

### Try It Out

Open the **AdvWorksDbContext.cs** file and modify the ***Database*** property to use the new SqlServerRepositoryBase class.

public SqlServerRepositoryBase Database

{ get; set; }

Open the **Program.cs** file and modify the code to look like the following.

string sql = "SalesLT.Product\_GetAllWithOutput";

ProductGetAllParam param = new() {

Result = ""

};

List<Product> list = db.Database

.SearchUsingStoredProcedure<Product,

ProductGetAllParam>(param, sql);

Add the following code after the loop displaying all the items returned.

Console.WriteLine();

Console.WriteLine($"Output Param:

'{param.Result}'");

Run the console application and you should see the OUTPUT parameter named **Result** appear after all the products have been displayed.

## Summary

This article built more functionality into the wrapper classes around ADO.NET to give you the ability to add WHERE clauses to SELECT statements. In addition, you saw how to retrieve data from views and stored procedures. Multiple result sets can be handled, and you can now retrieve scalar values. The best thing is that most of the code is going into generic classes, so as you add more classes to work with more tables, the code you write for each of those is minimal.

In the next article, you’ll learn to insert, update, and delete data. You will also learn to submit transactions, validate data using data annotations, and to handle exceptions.

Getting the Sample Code

You can download the sample code for this article by visiting www.CODEMag.com under the issue and article, or by visiting www.pdsa.com/downloads. Select "Articles" from the Category drop-down. Then select " Simplifying ADO.NET Code in .NET 6: Part 2" from the Item drop-down.

Sponsored Sidebar

Ready to Modernize a Legacy App?

Need FREE advice on migrating yesterday’s legacy applications to today’s modern platforms? Get answers by taking advantage of CODE Consulting’s years of experience by contacting us today to schedule your free hour of CODE consulting call. No strings. No commitment. Nothing to buy. For more information. visit [www.codemag.com/consulting](http://www.codemag.com/consulting) or email us at [info@codemag.com](mailto:info@codemag.com).

Paul D. Sheriff

http://www.pdsa.com

Paul has been in the IT industry over 35 years. In that time, he has successfully assisted hundreds of company's architect software applications to solve their toughest business problems. Paul has been a teacher and mentor through various mediums such as video courses, blogs, articles and speaking engagements at user groups and conferences around the world. Paul has many courses in the www.pluralsight.com library (<http://www.pluralsight.com/author/paul-sheriff>) on topics ranging from .NET 6, LINQ, JavaScript, Angular, MVC, WPF, ADO.NET, jQuery, and Bootstrap. Contact Paul at [psheriff@pdsa.com](mailto:psheriff@pdsa.com).

Listing 1: Add a base class for all the code that does not change between all repository classes

#nullable disable

using System.ComponentModel.DataAnnotations

.Schema;

using System.Data;

using System.Reflection;

using System.Text;

namespace AdoNetWrapper.Common;

public class RepositoryBase {

public RepositoryBase(

DatabaseContext context) {

DbContext = context;

Init();

}

protected readonly DatabaseContext DbContext;

public string SchemaName { get; set; }

public string TableName { get; set; }

public string SQL { get; set; }

public List<ColumnMapper> Columns { get; set; }

protected virtual void Init() {

SchemaName = "dbo";

TableName = string.Empty;

SQL = string.Empty;

Columns = new();

}

}

Listing 2: Modify the ProductRepository class to pass a Product object to the Search() method

#nullable disable

using AdoNetWrapper.Common;

using AdoNetWrapperSamples.Models;

using AdoNetWrapperSamples.EntityClasses;

namespace AdoNetWrapperSamples

.RepositoryClasses;

public class ProductRepository

: RepositoryBase {

public ProductRepository(

AdvWorksDbContext context)

: base(context) { }

public virtual List<Product> Search() {

return base.Search<Product>();

}

}

Listing 3: Create a DbContext class for each database you wish to interact with

#nullable disable

using AdoNetWrapper.Common;

using AdoNetWrapperSamples

.RepositoryClasses;

namespace AdoNetWrapperSamples.Models;

public partial class AdvWorksDbContext

: SqlServerDatabaseContext {

public AdvWorksDbContext(string connectString)

: base(connectString) { }

protected override void Init() {

base.Init();

Products = new(this);

}

public ProductRepository Products { get; set; }

}

Listing 4: Create method to build collection of properties for the search columns

protected virtual List<ColumnMapper>

BuildSearchColumnCollection<TEntity,

TSearch>(TSearch search) {

List<ColumnMapper> ret = new();

ColumnMapper colMap;

object value;

// Get all the properties in <TSearch>

PropertyInfo[] props =

typeof(TSearch).GetProperties();

// Loop through all properties

foreach (PropertyInfo prop in props) {

value = prop.GetValue(search, null);

// Is the search property filled in?

if (value != null) {

// Create a column mapping object

colMap = new() {

ColumnName = prop.Name,

PropertyInfo = prop,

SearchOperator = "=",

ParameterValue = value

};

// Does Property have a [Search] attribute

SearchAttribute sa = prop

.GetCustomAttribute<SearchAttribute>();

if (sa != null) {

// Set column name from [Search]

colMap.ColumnName =

string.IsNullOrWhiteSpace(sa.ColumnName)

? colMap.ColumnName : sa.ColumnName;

colMap.SearchOperator =

sa.SearchOperator ?? "=";

}

// Create collection of columns

ret.Add(colMap);

}

}

return ret;

}

Listing 5: Add a method to build a WHERE clause for searching

protected virtual string BuildSearchWhereClause

(List<ColumnMapper> columns) {

StringBuilder sb = new(1024);

string and = string.Empty;

// Create WHERE clause

sb.Append(" WHERE");

foreach (var item in columns) {

sb.Append($"{and} {item.ColumnName}

{item.SearchOperator}

{DbContext.ParameterPrefix}

{item.ColumnName}");

and = " AND";

}

return sb.ToString();

}

Listing 6: Add a method to build the parameters for the WHERE clause

protected virtual void BuildWhereClauseParameters

(IDbCommand cmd,

List<ColumnMapper> whereColumns) {

// Add parameters for each value passed in

foreach (ColumnMapper item in whereColumns) {

var param = DbContext.CreateParameter(

item.ColumnName,

item.SearchOperator == "LIKE" ?

item.ParameterValue + "%" :

item.ParameterValue);

cmd.Parameters.Add(param);

// Store parameter info

Columns.Find(c => c.ColumnName ==

item.ColumnName)

.ParameterValue = item.ParameterValue;

}

}

Listing 7: Add a Search() method that accepts a Command object

public virtual List<TEntity>

Search<TEntity>(IDbCommand cmd) {

List<TEntity> ret;

// Build Columns if needed

if (Columns.Count == 0) {

Columns = BuildColumnCollection<TEntity>();

}

// Set Command Object

DbContext.CommandObject = cmd;

// Get the list of entity objects

ret = BuildEntityList<TEntity>

(DbContext.CreateDataReader());

return ret;

}

Listing 8: Create an overloaded Search() method to accept a Product Search class

public virtual List<TEntity>

Search<TEntity, TSearch>(TSearch search) {

// Build SQL from Entity class

SQL = BuildSelectSql<TEntity>();

// Build collection of ColumnMapper objects

// from properties in the TSearch object

var searchColumns =

BuildSearchColumnCollection<TEntity,

TSearch>(search);

if (searchColumns != null &&

searchColumns.Any()) {

// Build the WHERE clause for Searching

SQL += BuildSearchWhereClause(searchColumns);

}

// Create Command Object with SQL

DbContext.CreateCommand(SQL);

// Add any Parameters?

if (searchColumns != null &&

searchColumns.Any()) {

BuildWhereClauseParameters(

DbContext.CommandObject, searchColumns);

}

return Search<TEntity>(DbContext.CommandObject);

}

Listing 9: Create a SQL statement and a Command object to submit a search

using AdvWorksDbContext db = new(ConnectString);

string sql = "SELECT \* FROM SalesLT.Product ";

sql += "WHERE Name LIKE @Name + '%'";

sql += " AND ListPrice >= @ListPrice";

// Create Command object

var cmd = db.CreateCommand(sql);

// Add Parameters

cmd.Parameters.Add(

db.CreateParameter("Name", "C"));

cmd.Parameters.Add(

db.CreateParameter("ListPrice", 50));

// Call the SELECT statement

List<Product> list =

db.Database.Search<Product>(cmd);

Console.WriteLine("\*\*\* Get Product Data \*\*\*");

// Display Data

foreach (var item in list) {

Console.WriteLine(item.ToString());

}

Console.WriteLine();

Console.WriteLine($"Total Items:

{list.Count}");

Console.WriteLine();

Listing 10: Add an Entity class to map the results returned from the view

#nullable disable

using System.ComponentModel

.DataAnnotations.Schema;

namespace AdoNetWrapperSamples.EntityClasses;

[Table("vProductAndDescription",

Schema = "SalesLT")]

public partial class ProductAndDescription {

public int ProductID { get; set; }

public string Name { get; set; }

public string ProductModel { get; set; }

public string Culture { get; set; }

public string Description { get; set; }

public override string ToString() {

return $"Name={Name} -

ProductModel={ProductModel} -

Description={Description}";

}

}

Listing 11: The Find() method retrieves a single entity from the table

public virtual TEntity Find<TEntity>

(params Object[] keyValues)

where TEntity : class {

// To assign null, use 'where TEntity : class'

TEntity ret = null;

if (keyValues != null) {

List<ColumnMapper> searchColumns;

// Build SQL from Entity class

SQL = BuildSelectSql<TEntity>();

// Build a collection of ColumnMapper

// objects based on [Key] attribute

searchColumns = Columns

.Where(col => col.IsKeyField).ToList();

// Number of [Key] attributes on entity class

// must match number of key values passed in

if (searchColumns.Count != keyValues.Length) {

throw new ApplicationException(

"Not enough parameters passed to Find()

method, or not enough [Key] attributes

on the entity class.");

}

// Set the values into the searchColumns

for (int i = 0; i < searchColumns.Count;

i++) {

searchColumns[i].ParameterValue =

keyValues[i];

searchColumns[i].SearchOperator = "=";

}

// Build the WHERE clause for Searching

SQL += BuildSearchWhereClause(searchColumns);

// Create command object with SQL

DbContext.CreateCommand(SQL);

// Add any Parameters?

if (searchColumns != null &&

searchColumns.Any()) {

BuildWhereClauseParameters(

DbContext.CommandObject, searchColumns);

}

// Get the entity

ret = Find<TEntity>(DbContext.CommandObject);

}

return ret;

}

Listing 12: The overload of the Find() method executes the command

public virtual TEntity

Find<TEntity>(IDbCommand cmd)

where TEntity : class {

// To assign null, use 'where TEntity : class'

TEntity ret = null;

// Build Columns if needed

if (Columns.Count == 0) {

Columns = BuildColumnCollection<TEntity>();

}

// Get the entity

var list = Search<TEntity>(cmd);

// Check for a single record

if (list != null && list.Any()) {

// Assign the object to the return value

ret = list[0];

}

return ret;

}

Listing 13: The Find() method returns a null if the record is not found, or it returns a valid entity object

using AdvWorksDbContext db = new(ConnectString);

Product entity = db.Products.**Find(706)**;

Console.WriteLine("\*\*\* Get Product Data \*\*\*");

**if (entity == null) {**

**Console.WriteLine(**

**"Can't Find Product ID=706");**

**}**

**else {**

**// Display Data**

**Console.WriteLine(entity.ToString());**

**Console.WriteLine();**

**Console.WriteLine(**

**$"SQL Submitted: {db.Products.SQL}");**

**}**

Console.WriteLine();

Listing 14: Add a new Search() method that takes an IDataReader object

public virtual List<TEntity>

Search<TEntity>(IDbCommand cmd,

IDataReader rdr) {

List<TEntity> ret;

// Build Columns if needed

if (Columns.Count == 0) {

Columns = BuildColumnCollection<TEntity>();

}

// Set Command Object

DbContext.CommandObject = cmd;

// Get the list of entity objects

ret = BuildEntityList<TEntity>(rdr);

return ret;

}

Listing 15: Add a new entity class to illustrate how to get multiple result sets

#nullable disable

using System.ComponentModel.DataAnnotations;

using System.ComponentModel

.DataAnnotations.Schema;

namespace AdoNetWrapperSamples.EntityClasses;

[Table("Customer", Schema = "SalesLT")]

public partial class Customer

{

[Key]

public int CustomerID { get; set; }

public string Title { get; set; }

public string FirstName { get; set; }

public string MiddleName { get; set; }

public string LastName { get; set; }

public string CompanyName { get; set; }

public override string ToString() {

return $"{LastName}, {FirstName}

({CustomerID})";

}

}

Listing 16: Create a class to wrap up both result sets

#nullable disable

using AdoNetWrapperSamples.EntityClasses;

using AdoNetWrapperSamples.Models;

namespace AdoNetWrapperSamples.ViewModelClasses;

public class ProductCustomerViewModel {

public ProductCustomerViewModel

(string connectString) {

ConnectString = connectString;

}

public string ConnectString { get; set; }

public List<Product> Products { get; set; }

public List<Customer> Customers { get; set; }

public void LoadProductsAndCustomers() {

string sql = "SELECT \*

FROM SalesLT.Product;";

sql += "SELECT \* FROM SalesLT.Customer";

using AdvWorksDbContext db =

new(ConnectString);

// Create Command object

var cmd = db.CreateCommand(sql);

// Get the Product Data

Products = db.Database.Search<Product>(cmd);

// Advance to next result set

db.DataReaderObject.NextResult();

// Clear columns to get ready

// for next result set

db.Database.Columns = new();

// Get the Customer Data

Customers = db.Database

.Search<Customer>(cmd, db.DataReaderObject);

}

}

Listing 17: Create a stored procedure to perform searching

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 IS NULL OR

Name LIKE @Name + '%')

AND (@ProductNumber IS NULL OR

ProductNumber LIKE @ProductNumber + '%')

AND (@BeginningCost IS NULL OR

StandardCost >= @BeginningCost)

AND (@EndingCost IS NULL OR

StandardCost <= @EndingCost)

END

Listing 18: Add new method to accept a SQL statement for calling a stored procedure

public virtual List<TEntity>

SearchUsingStoredProcedure<TEntity, TParam>

(TParam param, string sql) {

List<ColumnMapper> searchColumns = new();

List<TEntity> ret;

// Store the SQL submitted

SQL = sql;

// Build columns collection for entity class

Columns = BuildColumnCollection<TEntity>();

// Create Command Object with SQL

DbContext.CreateCommand(SQL);

// Set CommandType to Stored Procedure

DbContext.CommandObject.CommandType =

CommandType.StoredProcedure;

if (param!= null) {

// Build a collection of ColumnMapper objects

// based on properties in the TParam object

searchColumns = BuildSearchColumnCollection

<TEntity, TParam>( param);

// Add any Parameters?

if (searchColumns != null &&

searchColumns.Count > 0) {

BuildWhereClauseParameters(

DbContext.CommandObject, searchColumns);

}

}

ret = BuildEntityList<TEntity>

(DbContext.CreateDataReader());

return ret;

}

Listing 19: Modify the BuildWhereClauseParameters() method to set a DBNull.Value

protected virtual void

BuildWhereClauseParameters(IDbCommand cmd,

List<ColumnMapper> whereColumns) {

// Add parameters for each key value passed in

foreach (ColumnMapper item in whereColumns) {

var param = DbContext.CreateParameter(

item.ColumnName,

item.SearchOperator == "LIKE" ?

item.ParameterValue + "%" :

item.ParameterValue);

**// Add parameter value or DBNull value**

**param.Value ??= DBNull.Value;**

cmd.Parameters.Add(param);

**if (cmd.CommandType !=**

**CommandType.StoredProcedure) {**

// Store parameter info

Columns.Find(c => c.ColumnName ==

item.ColumnName)

.ParameterValue = item.ParameterValue;

**}**

}

}

Listing 20: Call a stored procedure using the SearchUsingStoredProcedure() method

using AdvWorksDbContext db = new(ConnectString);

string sql = "SalesLT.Product\_Search";

ProductSearchParam param = new() {

Name = "C"

};

List<Product> list = db.Database

.SearchUsingStoredProcedure<Product,

ProductSearchParam>(param, sql);

// Display Products

foreach (var item in list) {

Console.WriteLine(item);

}

Console.WriteLine();

Console.WriteLine(

$"Total Items: {list.Count}");

Console.WriteLine();

Console.WriteLine(

$"SQL Submitted: {db.Database.SQL}");

Listing 21: Create a new method to get the output parameter values

protected virtual void GetOutputParameters

<TParam>(TParam param,

List<ColumnMapper> columns) {

// Get output parameters

foreach (ColumnMapper item in columns

.Where(c => c.Direction ==

ParameterDirection.Output ||

c.Direction ==

ParameterDirection.InputOutput)) {

// Get the output parameter

var outParam = DbContext

.GetParameter(item.ColumnName);

// Set the value on the parameter object

typeof(TParam).GetProperty(item.ColumnName)

.SetValue(param, outParam.Value, null);

}

}

Listing 22: Create a SqlServerRepositoryBase class to override those methods that have SQL Server specific functionality

#nullable disable

using System.Data;

using System.Data.SqlClient;

namespace AdoNetWrapper.Common;

public class SqlServerRepositoryBase

: RepositoryBase {

public SqlServerRepositoryBase(

SqlServerDatabaseContext context)

: base(context) { }

protected override void

BuildOutputParameters(IDbCommand cmd,

List<ColumnMapper> columns) {

// Add output parameters

foreach (ColumnMapper item in columns

.Where(c => c.Direction ==

ParameterDirection.Output)) {

var param = (SqlParameter)DbContext

.CreateParameter(item.ColumnName, null);

param.Direction = item.Direction;

param.DbType = item.DbType;

// Need to set the Size for SQL Server

param.Size = item.Size;

cmd.Parameters.Add(param);

}

}

}

Listing 23: Modify the SearchUsingStoredProcedure() method to build output parameters

if (param != null) {

// Build collection of ColumnMapper objects

// based on properties in the TParam object

searchColumns = BuildSearchColumnCollection

<TEntity, TParam>(param);

// Add any Parameters?

if (searchColumns != null &&

searchColumns.Count > 0) {

BuildWhereClauseParameters(DbContext

.CommandObject, searchColumns);

}

**// Add any Output Parameters?**

**if (searchColumns.Where(c => c.Direction ==**

**ParameterDirection.Output ||**

**c.Direction ==**

**ParameterDirection.InputOutput).Any()) {**

**BuildOutputParameters(DbContext.CommandObject,**

**searchColumns);**

**}**

}