Simplify ADO.NET Using Generic Methods and Reflection

A common task we face as developers is to take data from a database table and create a collection of entity objects. We have several methods to accomplish this task. You can build a DataTable or DataReader, loop through the rows, build a new object for each row, and write lines of code to transfer column data to each corresponding property. Another method is to use an object relational mapper (ORM) such as the Entity Framework, Dapper or NHibernate which performs these operations for you. Have you ever wondered how these ORMs build the collection? Well, wonder no more. This blog post will show you how it is done.

The Product Entity Class

For this blog post, use the AdventureWorksLT sample database for SQL Server. There is a Product table in the SalesLT schema that contains a list of products. The goal is to take each row of data and transfer it into a Product object. When you are done, you have a generic list of Product objects that you can display in an HTML table or a ListControl in Windows Forms or WPF. First, you need to create a Product class to match to the columns in the SalesLT.Product table. The Product class should look like the following code.

```
public partial class Product : EFEntityBase
{
  public int ProductID { get; set; }
  public string Name { get; set; }
  public string ProductNumber { get; set; }
  public string Color { get; set; }
  public decimal StandardCost { get; set; }
  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; }
  public DateTime SellStartDate { get; set; }
  public DateTime? SellEndDate { get; set; }
  public DateTime? DiscontinuedDate { get; set; }
  public DateTime ModifiedDate { get; set; }
}
```

Read Data Using a SqlDataReader

The fastest method to read data from a database table is to use the SqlDataReader class. The ExecuteReader() method on a SqlCommand object returns a SqlDataReader object which is a fast, forward-only cursor. Add a method named GetProducts()to read a set of rows from a table in SQL Server and translate that data into a generic list of entity classes as shown in the code below.

Copyright © 2012-2019 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

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

  // Create a connection
  using (SqlConnection cnn =
           new SqlConnection(AppSettings.ConnectionString)) {
    // Create command object
    using (SqlCommand cmd =
             new SqlCommand("SELECT * FROM SalesLT.Product", cnn)) {
      // Open the connection
      cnn.Open();

      using (SqlDataReader dr =
          cmd.ExecuteReader(CommandBehavior.CloseConnection)) {
        while (dr.Read()) {
          ret.Add(new Product
          {
           // NOTE: GetFieldValue() does not work on nullable fields
            ProductID = dr.GetFieldValue<int>(
              dr.GetOrdinal("ProductID")),
            Name = dr.GetFieldValue<string>(
              dr.GetOrdinal("Name")),
            ProductNumber = dr.GetFieldValue<string>(
              dr.GetOrdinal("ProductNumber")),
            Color = dr.IsDBNull(dr.GetOrdinal("Color"))
              ? (string)null : Convert.ToString(dr["Color"]),
            StandardCost = dr.GetFieldValue<decimal>(
              dr.GetOrdinal("StandardCost")),
            ListPrice = dr.GetFieldValue<decimal>(
              dr.GetOrdinal("ListPrice")),
            Size = dr.IsDBNull(dr.GetOrdinal("Size"))
              ? (string)null : Convert.ToString(dr["Size"]),
            Weight = dr.IsDBNull(dr.GetOrdinal("Weight"))
              ? (decimal?)null : Convert.ToDecimal(dr["Weight"]),
            ProductCategoryID = dr.IsDBNull(
              dr.GetOrdinal("ProductCategoryID")) ? (int?)null
              : Convert.ToInt32(dr["ProductCategoryID"]),
            ProductModelID = dr.IsDBNull(
              dr.GetOrdinal("ProductModelID")) ? (int?)null
              : Convert.ToInt32(dr["ProductModelID"]),
            SellStartDate = dr.GetFieldValue<DateTime>(
              dr.GetOrdinal("SellStartDate")),
            SellEndDate = dr.IsDBNull(dr.GetOrdinal("SellEndDate"))
              ? (DateTime?)null
              : Convert.ToDateTime(dr["SellEndDate"]),
            DiscontinuedDate = dr.IsDBNull(
              dr.GetOrdinal("DiscontinuedDate")) ? (DateTime?)null
              : Convert.ToDateTime(dr["DiscontinuedDate"]),
            ModifiedDate = dr.GetFieldValue<DateTime>(
              dr.GetOrdinal("ModifiedDate")),
          });
        }
      }
    }
  }
```


Copyright © 2012-2019 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

```
  return ret;
}
```

The code above creates a SqlConnection object and a SqlCommand object in order to submit a SELECT statement to the AdventureWorksLT database and read all rows and columns from the Product table. The ExecuteReader() method returns a fast, forward-only cursor in the form of a SqlDataReader. Loop through each row using the Read() method and each time through the loop, add a new Product object to a generic List<Product> collection.

When you read each column from the data reader, you need to know if the property is defined as a nullable type. If it is not a nullable type, call the GetFieldValue() method to read the value in the field as shown below.

```
ProductID = dr.GetFieldValue<int>(dr.GetOrdinal("ProductID"))
```

If the field is a nullable type, check for a null value and either put a null value into the field, or read the data and convert it into the appropriate type as shown in the following code snippet.

```
Color = dr.IsDBNull(dr.GetOrdinal("Color"))
            ? (string)null
            : Convert.ToString(dr["Color"])
```

When there are no more rows, the data reader is closed and disposed of because it is wrapped into a using() block. The command and connection objects are also closed and disposed of because of being wrapped into using() blocks.

While the code shown above is not too much, think about if you need to write this code for every table in your database. Also, think about the amount of code you need to write and maintain each time a table is added, deleted, or changed. You can see that maintaining all this code can get overwhelming quickly. You can simplify the above code by using some generic methods and a little bit of .NET reflection.

Read Data Using Reflection

Instead of writing the code to loop through the rows in the data reader and assigning the data in each column to the corresponding properties, you can make this code generic. The method below is the same as the one above, but instead of using all the code to take each column and transfer the data into each property, that code is made generic and moved into a ToList<T>() method. The ToList() method reads the columns from the DataReader and assumes the property names are the same name. Reflection is used to populate the properties with the column data.

Copyright © 2012-2019 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

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

  // Create a connection
  using (SqlConnection cnn =
           new SqlConnection(AppSettings.ConnectionString)) {
    // Create command object
    using (SqlCommand cmd =
        new SqlCommand("SELECT * FROM SalesLT.Product", cnn)) {
      // Open the connection
      cnn.Open();

      using (SqlDataReader dr =
          cmd.ExecuteReader(CommandBehavior.CloseConnection)) {
        ret = ToList<Product>(dr);
      }
    }
  }

  return ret;
}
```

ToList() Method

The ToList() method is a generic method that accepts any type of class you wish to fill with data from the DataReader object. The ToList() method calls the GetProperties() method on the Type class to get an array of PropertyInfo objects. Each PropertyInfo object contains information about each individual property in the Product class. Next, the code loops through all fields in the DataReader and determines if each field name is contained in the array of PropertyInfo objects. If the property is found, that PropertyInfo object is added to a generic list of PropertyInfo objects. This new list now just contains the list of columns to match to properties when looping through each row.

Loop through all rows in the DataReader using the Read() method. Create a new instance of the class passed in as T to this generic method using the CreateInstance() method of the Activator class. Loop through the list of columns found to match the properties and check to see if the data in that column contains a null or not. If the value is a null, set the property value to null, otherwise, read the column data and assign that to the property. Use the SetValue() property on the PropertyInfo object to assign the data as this is the fastest reflection method.

Copyright © 2012-2019 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

```
public virtual List<T> ToList<T>(IDataReader rdr)
{
  List<T> ret = new List<T>();
  T entity;
  Type typ = typeof(T);
  PropertyInfo col;
  List<PropertyInfo> columns = new List<PropertyInfo>();

  // Get all the properties in Entity Class
  PropertyInfo[] props = typ.GetProperties();

  // Loop through one time to map columns to properties
  // NOTES:
  //   Assumes your column names are the same name
  //   as your class property names
  //   Any properties not in the data reader column list are not set
  for (int index = 0; index < rdr.FieldCount; index++) {
    // See if column name maps directly to property name
    col = props.FirstOrDefault(c => c.Name == rdr.GetName(index));
    if (col != null) {
      columns.Add(col);
    }
  }

  // Loop through all records
  while (rdr.Read()) {
    // Create new instance of Entity
    entity = Activator.CreateInstance<T>();

    // Loop through columns to assign data
    for (int i = 0; i < columns.Count; i++) {
      if (rdr[columns[i].Name].Equals(DBNull.Value)) {
        columns[i].SetValue(entity, null, null);
      }
      else {
        columns[i].SetValue(entity, rdr[columns[i].Name], null);
      }
    }

    ret.Add(entity);
  }

  return ret;
}
```

NOTE: The ToList<T>() method assumes the property names and your column names in the table are the same. If you wish to use different names, then you need to add code to use the [Column] attribute.

Copyright © 2012-2019 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.

Read Data Using Generic Method

The code to create a SqlConnection and SqlCommand objects would be the same for any table you wish to read. The only things that change are the SQL string, the connection string, and the type of object to create. Thus, you can simplify the GetProducts() method even further by creating a GetRecords<T>() method that accepts the class to create a list of, the SQL string, and the connection string.

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

  ret =
    GetRecords<Product>("SELECT * FROM SalesLT.Product",
                        AppSettings.ConnectionString);

  return ret;
}
```

GetRecords() Method

The GetRecords() method accepts the SQL string and connection string. It creates a List<T> collection variable called *ret*. It then builds the connection and command objects using the parameters passed in. It opens the connection and creates the SqlDataReader object. It calls the ToList<T>() method you looked at earlier to build the generic collection of objects. This collection is then returned from this method.

```
public List<T> GetRecords<T>(string sql, string connectionString)
{
  List<T> ret = new List<T>();

  // Create a connection
  using (SqlConnection cnn = new SqlConnection(connectionString)) {
    // Create command object
    using (SqlCommand cmd = new SqlCommand(sql, cnn)) {
      // Open the connection
      cnn.Open();

      using (SqlDataReader dr =
         cmd.ExecuteReader(CommandBehavior.CloseConnection)) {
        ret = ToList<T>(dr);
      }
    }
  }

  return ret;
}
```


Copyright © 2012-2019 by Paul D. Sheriff
All rights reserved. Reproduction is strictly prohibited.

Summary

In this blog post, you learned to simplify ADO.NET code by using some generic methods and reflection. This code is very similar to the way most ORMs populate data under the hood. Since you are using a SqlDataReader to get the data, and reading the properties to populate one time, this code is very efficient. In fact, if you did some timings, this code should be faster than EF, Dapper, and NHibernate since there is none of the overhead these ORMs add.

Source Code

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Fairway/PDSA Blog,” then select “Simplify ADO.NET Using Generic Methods and Reflection” from the dropdown list.

Copyright © 2012-2019 by Paul D. Sheriff
All rights reserved worldwide. Reproduction is strictly prohibited.