Creating Collections of Entity Objects using Reflection

In my last blog posts I have been showing you how to create collection of entity objects using code that is custom for each table and object you create. Well, if you use a little reflection code you can shrink this code quite a bit. Yes, we all know that reflection is slow and probably should be avoided in most cases. What I have found out is that loading over 6200 product records into an entity collection still takes less than a second when using Reflection. So, I will leave it up to you to decide which way you wish to go.

We will once again use our Product class that uses nullable types as shown below:

```
C#
public class Product
{
  public int? ProductId { get; set; }
  public string ProductName { get; set; }
  public DateTime? IntroductionDate { get; set; }
  public decimal? Cost { get; set; }
  public decimal? Price { get; set; }
  public bool? IsDiscontinued { get; set; }
}

Visual Basic
Public Class Product
  Public Property ProductId() AsNullable(Of Integer)
  Public Property ProductName() As String
  Public Property IntroductionDate() As Nullable(Of DateTime)
  Public Property Cost() As Nullable(Of Decimal)
  Public Property Price() As Nullable(Of Decimal)
  Public Property IsDiscontinued() As Nullable(Of Boolean)
End Class
```

How Reflection Works

If you wish to set one of the properties on the Product class to a certain value, you write code like the following:

```
C#
Product entity = new Product();
entity.ProductName = "A New Product";

Visual Basic
Dim entity as New Product()
entity.ProductName = "A New Product"
```

Sometimes you might want to create a generic routine that you can pass a property name to and the value to set that property to. This can be accomplished using Reflection as shown in the following code:

```
C#
Product entity = new Product();
typeof(Product).InvokeMember("ProductName",
  BindingFlags.SetProperty,
    Type.DefaultBinder, entity,
     new Object[] { "A New Product" });


Visual Basic
Dim entity as New Product()
GetType(Product).InvokeMember("ProductName", _
   BindingFlags.SetProperty, _
     Type.DefaultBinder, entity, _
        New Object() { "A New Product" })
```

The InvokeMember is a method of the System.Type class. Using typeof() in C# or GetType() in Visual Basic returns an instance of the Type class which contains meta-data about the Product class. You pass 5 parameters to the InvokeMember method. The first parameter is the name of the property you wish to set. The second parameter is the name of the property or method you wish to invoke; in this case it is the Set property. The third parameter tells InvokeMember that you are using the default binder. The fourth parameter is the variable that contains a reference to an instance of the class specified by the type (in this case the Product object). The last parameter is an object array of whatever you need to pass to the method or property that you are invoking. For setting the ProductName property you only need a single object array of the string you are setting.

A Better Way to Set Property Values

While the InvokeMember method works for setting a property, it is actually quite slow. There is a more efficient way to set a property using reflection. There is a GetProperty method on the Type class you use to retrieve a PropertyInfo object. This PropertyInfo object has a SetValue method that you can use to set the value on that property. Below is an example of calling the SetValue method.

```
C#
Product entity = new Product();

typeof(Product).GetProperty("ProductName").
  SetValue(entity, "A New Product", null);

MessageBox.Show(entity.ProductName);


Visual Basic
Dim entity As New Product()

GetType(Product).GetProperty("ProductName"). _
  SetValue(entity, "A New Product", Nothing)

MessageBox.Show(entity.ProductName)
```

The above code is actually a little easier to understand than using the InvokeMember and is over 100% faster! That is a big difference and you should take advantage of it!

Apply Reflection to Loading Collections

When you wish to load a collection of entity classes you will loop through either a DataReader or a DataTable. Before you loop through, however, you should gather a collection of all properties on your Product class into an array of PropertyInfo objects. This way you only get the properties one time instead of each time through the rows you get a single property using the GetProperty method. In the code shown below you will use the GetProperties method to retrieve this array.

You will then build the data reader and move through each row of the data reader by using the Read method. For each row you will now loop through the PropertyInfo array and use the property name to retrieve the corresponding column in the data reader. Remember, this assumes that your column names are the same name as your entity class.

```
C#
public List<Product> GetProducts()
{
  SqlCommand cmd = null;
  List<Product> ret = new List<Product>();
  Product entity = null;

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

  cmd = new SqlCommand("SELECT * FROM Product");
  using (cmd.Connection = new
          SqlConnection(AppSettings.Instance.ConnectString))
  {
    cmd.Connection.Open();
    using (var rdr = cmd.ExecuteReader())
    {
      while (rdr.Read())
      {
        // Create new instance of Product Class
        entity = new Product();

        // Set all properties from the column names
        // NOTE: This assumes your column names are the
        //       same name as your class property names
        foreach (PropertyInfo col in props)
        {
          if (rdr[col.Name].Equals(DBNull.Value))
            col.SetValue(entity, null, null);
          else
            col.SetValue(entity, rdr[col.Name], null);
        }

        ret.Add(entity);
      }
    }
  }

  return ret;
}


Visual Basic
Public Function GetProducts() As List(Of Product)
  Dim cmd As SqlCommand = Nothing
  Dim ret As New List(Of Product)()
  Dim entity As Product = Nothing

  ' Get all the properties in Entity Class
  Dim props As PropertyInfo() = _
      GetType(Product).GetProperties()

  cmd = New SqlCommand("SELECT * FROM Product")
  Using cnn = New _
          SqlConnection(AppSettings.Instance.ConnectString)
    cmd.Connection = cnn
    cmd.Connection.Open()
    Using rdr = cmd.ExecuteReader()
      While rdr.Read()
        ' Create new instance of Product Class
        entity = New Product()

        ' Set all properties from the column names
        ' NOTE: This assumes your column names are the
        '       same name as your class property names
        For Each col As PropertyInfo In props
          If rdr(col.Name).Equals(DBNull.Value) Then
            col.SetValue(entity, Nothing, Nothing)
          Else
            col.SetValue(entity, rdr(col.Name), Nothing)
          End If
        Next

        ret.Add(entity)
      End While
    End Using
  End Using

  Return ret
End Function
```

Create Generic Base Class

Instead of writing all of the above code for each entity collection class you need to load, you can create a base class with a generic method that will build your collection for you. Create a class called ManagerBase to which you will create a method called BuildCollection. This BuildCollection method will allow you to specify the type of entity, symbolized by <T>, that you wish to create a collection of. Pass into this method the type of the entity and a SqlDataReader and this method will take care of the rest. With the entity Type you pass in this method can retrieve the array of PropertyInfo objects from that type. A loop is made through the data reader and a new instance of the entity is created using the Activator class' CreateInstance method. All the properties in the array of PropertyInfo objects is looped through to gather the data into the entity. Each entity is finally added to a generic List<T> collection. When all records have been processed the generic list is returned.

```
C#
public class ManagerBase
{
  public List<T> BuildCollection<T>(Type typ,
     SqlDataReader rdr)
  {
    List<T> ret = new List<T>();
    T entity;

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

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

      // Set all properties from the column names
      // NOTE: This assumes your column names are the
      //       same name as your class property names
      foreach (PropertyInfo col in props)
      {
        if (rdr[col.Name].Equals(DBNull.Value))
          col.SetValue(entity, null, null);
        else
          col.SetValue(entity, rdr[col.Name], null);
      }

      ret.Add(entity);
    }

    return ret;
  }
}


Visual Basic
Public Class ManagerBase
  Public Function BuildCollection(Of T)(typ As Type, _
    rdr As SqlDataReader) As List(Of T)
    Dim ret As New List(Of T)()
    Dim entity As T

    ' Get all the properties in Entity Class
    Dim props As PropertyInfo() = typ.GetProperties()

    While rdr.Read()
      ' Create new instance of Entity
      entity = Activator.CreateInstance(Of T)()

      ' Set all properties from the column names
      ' NOTE: This assumes your column names are the
      '       same name as your class property names
      For Each col As PropertyInfo In props
        If rdr(col.Name).Equals(DBNull.Value) Then
          col.SetValue(entity, Nothing, Nothing)
        Else
          col.SetValue(entity, rdr(col.Name), Nothing)
        End If
      Next

      ret.Add(entity)
    End While

    Return ret
  End Function
End Class
```

Use Base Class

To use this base class you will create your ProductManager class that inherits from this ManagerBase class. You can rewrite the GetProducts method shown above with the code shown below. You can see that this significantly reduces the amount of code you need to write.

```
C#
public class ProductManager : ManagerBase
{
  public List<Product> GetProducts()
  {
    SqlCommand cmd = null;
    List<Product> ret = null;

    cmd = new SqlCommand("SELECT * FROM Product");
    using (cmd.Connection = new
            SqlConnection(AppSettings.Instance.ConnectString))
    {
      cmd.Connection.Open();
      using (var rdr = cmd.ExecuteReader())
      {
        // Build Collection of Entity Objets using Reflection
        ret = BuildCollection<Product>(typeof(Product), rdr);
      }
    }

    return ret;
  }
}

Visual Basic
Public Class ProductManager
  Inherits ManagerBase

  Public Function GetProducts() As List(Of Product)
    Dim cmd As SqlCommand = Nothing
    Dim ret As List(Of Product) = Nothing

    cmd = New SqlCommand("SELECT * FROM Product")
    Using cnn = New _
       SqlConnection(AppSettings.Instance.ConnectString)
      cmd.Connection = cnn
      cmd.Connection.Open()
      Using rdr = cmd.ExecuteReader()

        ' Build Collection of Entity Objets using Reflection
        ret = BuildCollection(Of Product)( _
                  GetType(Product), rdr)

      End Using
    End Using

    Return ret
  End Function
End Class
```

Summary

In this blog post you learned how to use reflection to fill a collection of entity objects. There are two different methods of setting properties using Reflection. You should use the SetValue method instead of the InvokeMember as it is more efficient. Creating a base class and using a generic method will eliminate a lot of repetitive code.

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Tips & Tricks”, then select “Creating Collections of Entities using Reflection” from the drop down list.