This tutorial is part of a set. Find out more about data access with ASP.NET in the Working with Data in ASP.NET 2.0 section of the ASP.NET site at http://www.asp.net/learn/dataaccess/default.aspx.

Working with Data in ASP.NET 2.0 :: Creating a Business Logic Layer

Introduction

The Data Access Layer (DAL) created in the first tutorial cleanly separates the data access logic from the presentation logic. However, while the DAL cleanly separates the data access details from the presentation layer, it does not enforce any business rules that may apply. For example, for our application we may want to disallow the `CategoryID` or `SupplierID` fields of the `Products` table to be modified when the `Discontinued` field is set to 1, or we might want to enforce seniority rules, prohibiting situations in which an employee is managed by someone who was hired after them. Another common scenario is authorization – perhaps only users in a particular role can delete products or can change the `UnitPrice` value.

In this tutorial we'll see how to centralize these business rules into a Business Logic Layer (BLL) that serves as an intermediary for data exchange between the presentation layer and the DAL. In a real-world application, the BLL should be implemented as a separate Class Library project; however, for these tutorials we'll implement the BLL as a series of classes in our `App_Code` folder in order to simplify the project structure. Figure 1 illustrates the architectural relationships among the presentation layer, BLL, and DAL.

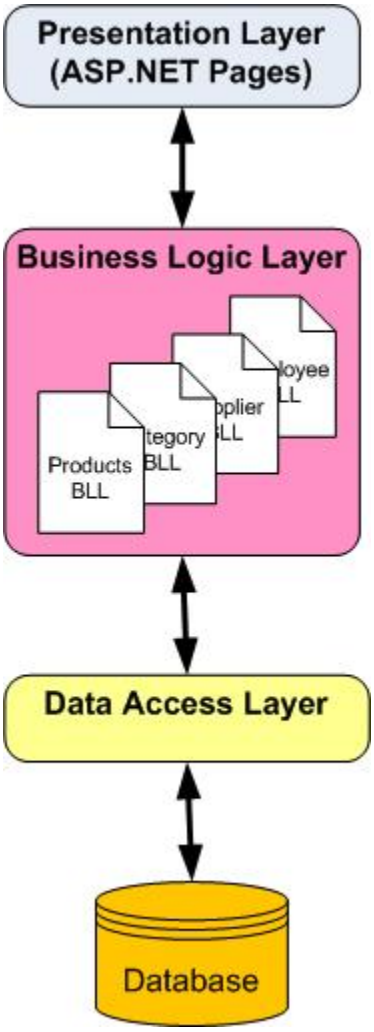

Figure 1: The BLL Separates the Presentation Layer from the Data Access Layer and Imposes Business Rules

Rather than creating separate classes to implement our business logic, we could alternatively place this logic directly in the Typed DataSet with partial classes. For an example of creating and extending a Typed DataSet, refer back to the first tutorial.

Step 1: Creating the BLL Classes

Our BLL will be composed of four classes, one for each TableAdapter in the DAL; each of these BLL classes will have methods for retrieving, inserting, updating, and deleting from the respective TableAdapter in the DAL, applying the appropriate business rules.

To more cleanly separate the DAL- and BLL-related classes, let's create two subfolders in the `App_Code` folder, `DAL` and `BLL`. Simply right-click on the `App_Code` folder in the Solution Explorer and choose New Folder. After creating these two folders, move the Typed DataSet created in the first tutorial into the `DAL` subfolder.

Next, create the four BLL class files in the `BLL` subfolder. To accomplish this, right-click on the `BLL` subfolder, choose Add a New Item, and choose the Class template. Name the four classes `ProductsBLL`, `CategoriesBLL`, `SuppliersBLL`, and `EmployeesBLL`.

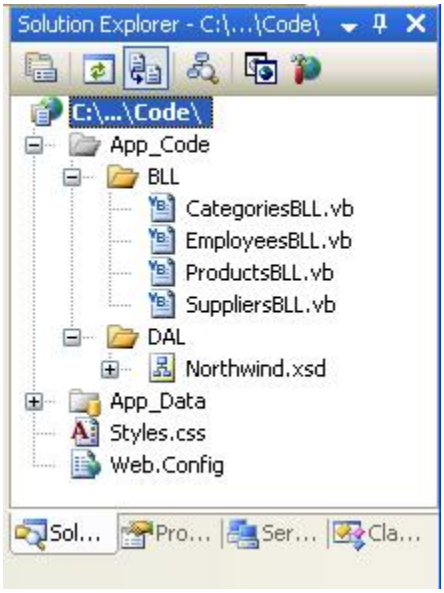

Figure 2: Add Four New Classes to the `App_Code` Folder

Next, let's add methods to each of the classes to simply wrap the methods defined for the TableAdapters from the first tutorial. For now, these methods will just call directly into the DAL; we'll return later to add any needed business logic.

Note: If you are using Visual Studio Standard Edition or above (that is, you're *not* using Visual Web Developer), you can optionally design your classes visually using the Class Designer. Refer to the Class Designer Blog for more information on this new feature in Visual Studio.

For the `ProductsBLL` class we need to add a total of seven methods:

- `GetProducts()` – returns all products
- `GetProductByProductID(`*`productID`*`)` – returns the product with the specified product ID
- `GetProductsByCategoryID(`*`categoryID`*`)` – returns all products from the specified category
- `GetProductsBySupplier(`*`supplierID`*`)` – returns all products from the specified supplier
- `AddProduct(`*`productName, supplierID, categoryID, quantityPerUnit, unitPrice, unitsInStock, unitsOnOrder, reorderLevel, discontinued`*`)` – inserts a new product into the database using the values passed-in; returns the `ProductID` value of the newly inserted record
- `UpdateProduct(`*`productName, supplierID, categoryID, quantityPerUnit, unitPrice, unitsInStock, unitsOnOrder, reorderLevel, discontinued, productID`*`)` – updates an existing product in the database using the passed-in values; returns `True` if precisely one row was updated, `False` otherwise
- `DeleteProduct(`*`productID`*`)` – deletes the specified product from the database

ProductsBLL.vb

```
Imports NorthwindTableAdapters

<System.ComponentModel.DataObject()> _
Public Class ProductsBLL

    Private _productsAdapter As ProductsTableAdapter = Nothing
    Protected ReadOnly Property Adapter() As ProductsTableAdapter
        Get
            If _productsAdapter Is Nothing Then
                _productsAdapter = New ProductsTableAdapter()
```

```
        End If

        Return _productsAdapter
    End Get
End Property

<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, True)> _
Public Function GetProducts() As Northwind.ProductsDataTable
    Return Adapter.GetProducts()
End Function

<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetProductByProductID(ByVal productID As Integer) _
    As Northwind.ProductsDataTable
    Return Adapter.GetProductByProductID(productID)
End Function

<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetProductsByCategoryID(ByVal categoryID As Integer) _
    As Northwind.ProductsDataTable
    Return Adapter.GetProductsByCategoryID(categoryID)
End Function

<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Select, False)> _
Public Function GetProductsBySupplierID(ByVal supplierID As Integer) _
    As Northwind.ProductsDataTable
    Return Adapter.GetProductsBySupplierID(supplierID)
End Function


<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Insert, True)> _
Public Function AddProduct( _
    productName As String, supplierID As Nullable(Of Integer), _
    categoryID As Nullable(Of Integer), quantityPerUnit As String, _
    unitPrice As Nullable(Of Decimal), unitsInStock As Nullable(Of Short), _
    unitsOnOrder As Nullable(Of Short), reorderLevel As Nullable(Of Short), _
    discontinued As Boolean) _
    As Boolean

    Dim products As New Northwind.ProductsDataTable()
    Dim product As Northwind.ProductsRow = products.NewProductsRow()

    product.ProductName = productName
    If Not supplierID.HasValue Then
        product.SetSupplierIDNull()
    Else
        product.SupplierID = supplierID.Value
    End If

    If Not categoryID.HasValue Then
        product.SetCategoryIDNull()
    Else
        product.CategoryID = categoryID.Value
    End If

    If quantityPerUnit Is Nothing Then
        product.SetQuantityPerUnitNull()
    Else
        product.QuantityPerUnit = quantityPerUnit
    End If
```

```
    If Not unitPrice.HasValue Then
        product.SetUnitPriceNull()
    Else
        product.UnitPrice = unitPrice.Value
    End If

    If Not unitsInStock.HasValue Then
        product.SetUnitsInStockNull()
    Else
        product.UnitsInStock = unitsInStock.Value
    End If

    If Not unitsOnOrder.HasValue Then
        product.SetUnitsOnOrderNull()
    Else
        product.UnitsOnOrder = unitsOnOrder.Value
    End If

    If Not reorderLevel.HasValue Then
        product.SetReorderLevelNull()
    Else
        product.ReorderLevel = reorderLevel.Value
    End If

    product.Discontinued = discontinued


    products.AddProductsRow(product)
    Dim rowsAffected As Integer = Adapter.Update(products)

    Return rowsAffected = 1
End Function

<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Update, True)> _
Public Function UpdateProduct(_
    productName As String, supplierID As Nullable(Of Integer), _
    categoryID As Nullable(Of Integer), quantityPerUnit As String, _
    unitPrice As Nullable(Of Decimal), unitsInStock As Nullable(Of Short), _
    unitsOnOrder As Nullable(Of Short), reorderLevel As Nullable(Of Short), _
    discontinued As Boolean, productID As Integer) _
    As Boolean

    Dim products As Northwind.ProductsDataTable = _
        Adapter.GetProductByProductID(productID)

    If products.Count = 0 Then
        Return False
    End If

    Dim product as Northwind.ProductsRow = products(0)

    product.ProductName = productName
    If Not supplierID.HasValue Then
        product.SetSupplierIDNull()
    Else
        product.SupplierID = supplierID.Value
    End If

    If Not categoryID.HasValue Then
        product.SetCategoryIDNull()
    Else
        product.CategoryID = categoryID.Value
    End If
```

```
        If quantityPerUnit Is Nothing Then
            product.SetQuantityPerUnitNull()
        Else
            product.QuantityPerUnit = quantityPerUnit
        End If

        If Not unitPrice.HasValue Then
            product.SetUnitPriceNull()
        Else
            product.UnitPrice = unitPrice.Value
        End If

        If Not unitsInStock.HasValue Then
            product.SetUnitsInStockNull()
        Else
            product.UnitsInStock = unitsInStock.Value
        End If

        If Not unitsOnOrder.HasValue Then
            product.SetUnitsOnOrderNull()
        Else
            product.UnitsOnOrder = unitsOnOrder.Value
        End If

        If Not reorderLevel.HasValue Then
            product.SetReorderLevelNull()
        Else
            product.ReorderLevel = reorderLevel.Value
        End If

        product.Discontinued = discontinued

        Dim rowsAffected As Integer = Adapter.Update(product)

        Return rowsAffected = 1
    End Function

    <System.ComponentModel.DataObjectMethodAttribute _
        (System.ComponentModel.DataObjectMethodType.Delete, True)> _
    Public Function DeleteProduct(ByVal productID As Integer) As Boolean
        Dim rowsAffected As Integer = Adapter.Delete(productID)

        Return rowsAffected = 1
    End Function
End Class
```

The methods that simply return data – `GetProducts`, `GetProductByProductID`, `GetProductsByCategoryID`, and `GetProductBySuppliersID` – are fairly straightforward as they simply call down into the DAL. While in some scenarios there may be business rules that need to be implemented at this level (such as authorization rules based on the currently logged on user or the role to which the user belongs), we'll simply leave these methods as-is. For these methods, then, the BLL serves merely as a proxy through which the presentation layer accesses the underlying data from the Data Access Layer.

The `AddProduct` and `UpdateProduct` methods both take in as parameters the values for the various product fields and add a new product or update an existing one, respectively. Since many of the `Product` table's columns can accept `NULL` values (`CategoryID`, `SupplierID`, and `UnitPrice`, to name a few), those input parameters for `AddProduct` and `UpdateProduct` that map to such columns use nullable types. Nullable types are new to .NET 2.0 and provide a technique for indicating whether a value type should, instead, be `Nothing`. Refer to the Paul Vick's blog entry The Truth About Nullable Types and VB and the technical documentation for the Nullable structure for more information.

All three methods return a Boolean value indicating whether a row was inserted, updated, or deleted since the operation may not result in an affected row. For example, if the page developer calls `DeleteProduct` passing in a `ProductID` for a non-existent product, the `DELETE` statement issued to the database will have no affect and therefore the `DeleteProduct` method will return `False`.

Note that when adding a new product or updating an existing one we take in the new or modified product's field values as a list of scalars as opposed to accepting a `ProductsRow` instance. This approach was chosen because the `ProductsRow` class derives from the ADO.NET `DataRow` class, which doesn't have a default parameterless constructor. In order to create a new `ProductsRow` instance, we must first create a `ProductsDataTable` instance and then invoke its `NewProductRow()` method (which we do in `AddProduct`). This shortcoming rears its head when we turn to inserting and updating products using the ObjectDataSource. In short, the ObjectDataSource will try to create an instance of the input parameters. If the BLL method expects a `ProductsRow` instance, the ObjectDataSource will try to create one, but fail due to the lack of a default parameterless constructor. For more information on this problem, refer to the following two ASP.NET Forums posts: Updating ObjectDataSources with Strongly-Typed DataSets, and Problem With ObjectDataSource and Strongly-Typed DataSet.

Next, in both `AddProduct` and `UpdateProduct`, the code creates a `ProductsRow` instance and populates it with the values just passed in. When assigning values to DataColumns of a DataRow various field-level validation checks can occur. Therefore, manually putting the passed in values back into a DataRow helps ensure the validity of the data being passed to the BLL method. Unfortunately the strongly-typed DataRow classes generated by Visual Studio do not use nullable types. Rather, to indicate that a particular DataColumn in a DataRow should correspond to a `NULL` database value we must use the `Set`*`ColumnName`*`Null()` method.

In `UpdateProduct` we first load in the product to update using `GetProductByProductID(`*`productID`*`)`. While this may seem like an unnecessary trip to the database, this extra trip will prove worthwhile in future tutorials that explore optimistic concurrency. Optimistic concurrency is a technique to ensure that two users who are simultaneously working on the same data don't accidentally overwrite one another's changes. Grabbing the entire record also makes it easier to create update methods in the BLL that only modify a subset of the DataRow's columns. When we explore the `SuppliersBLL` class we'll see such an example.

Finally, note that the `ProductsBLL` class has the DataObject attribute applied to it (the `[System.ComponentModel.DataObject]` syntax right before the class statement near the top of the file) and the methods have DataObjectMethodAttribute attributes. The `DataObject` attribute marks the class as being an object suitable for binding to an ObjectDataSource control, whereas the `DataObjectMethodAttribute` indicates the purpose of the method. As we'll see in future tutorials, ASP.NET 2.0's ObjectDataSource makes it easy to declaratively access data from a class. To help filter the list of possible classes to bind to in the ObjectDataSource's wizard, by default only those classes marked as `DataObjects` are shown in the wizard's drop-down list. The `ProductsBLL` class will work just as well without these attributes, but adding them makes it easier to work with in the ObjectDataSource's wizard.

Adding the Other Classes

With the `ProductsBLL` class complete, we still need to add the classes for working with categories, suppliers, and employees. Take a moment to create the following classes and methods using the concepts from the example above:

- **CategoriesBLL.cs**
  - `GetCategories()`
  - `GetCategoryByCategoryID(`*`categoryID`*`)`
- **SuppliersBLL.cs**

    - GetSuppliers()
    - GetSupplierBySupplierID(*supplierID*)
    - GetSuppliersByCountry(*country*)
    - UpdateSupplierAddress(*supplierID*, *address*, *city*, *country*)
- **EmployeesBLL.cs**
    - GetEmployees()
    - GetEmployeeByEmployeeID(*employeeID*)
    - GetEmployeesByManager(*managerID*)

The one method worth noting is the `SuppliersBLL` class's `UpdateSupplierAddress` method. This method provides an interface for updating just the supplier's address information. Internally, this method reads in the `SupplierDataRow` object for the specified `supplierID` (using `GetSupplierBySupplierID`), sets its address-related properties, and then calls down into the `SupplierDataTable`'s `Update` method. The `UpdateSupplierAddress` method follows:

```
<System.ComponentModel.DataObjectMethodAttribute _
    (System.ComponentModel.DataObjectMethodType.Update, True)> _
Public Function UpdateSupplierAddress(ByVal supplierID As Integer, _
    ByVal address As String, ByVal city As String, ByVal country As String) _
    As Boolean

    Dim suppliers As Northwind.SuppliersDataTable = _
        Adapter.GetSupplierBySupplierID(supplierID)

    If suppliers.Count = 0 Then
        Return False
    Else
        Dim supplier As Northwind.SuppliersRow = suppliers(0)

        If address Is Nothing Then
            supplier.SetAddressNull()
        Else
            supplier.Address = address
        End If

        If city Is Nothing Then
            supplier.SetCityNull()
        Else
            supplier.City = city
        End If

        If country Is Nothing Then
            supplier.SetCountryNull()
        Else
            supplier.Country = country
        End If

        Dim rowsAffected As Integer = Adapter.Update(supplier)

        Return rowsAffected = 1
    End If
End Function
```

Refer to this article's download for my complete implementation of the BLL classes.

Step 2: Accessing the Typed DataSets Through the BLL Classes

In the first tutorial we saw examples of working directly with the Typed DataSet programmatically, but with the addition of our BLL classes, the presentation tier should work against the BLL instead. In the `AllProducts.aspx` example from the first tutorial, the `ProductsTableAdapter` was used to bind the list of products to a GridView, as shown in the following code:

```
Dim productsAdapter As New ProductsTableAdapter()
GridView1.DataSource = productsAdapter.GetProducts()
GridView1.DataBind()
```

To use the new BLL classes, all that needs to be changed is the first line of code – simply replace the `ProductsTableAdapter` object with a `ProductBLL` object:

```
Dim productLogic As New ProductsBLL()
GridView1.DataSource = productLogic.GetProducts()
GridView1.DataBind()
```

The BLL classes can also be accessed declaratively (as can the Typed DataSet) by using the ObjectDataSource. We'll be discussing the ObjectDataSource in greater detail in the following tutorials.

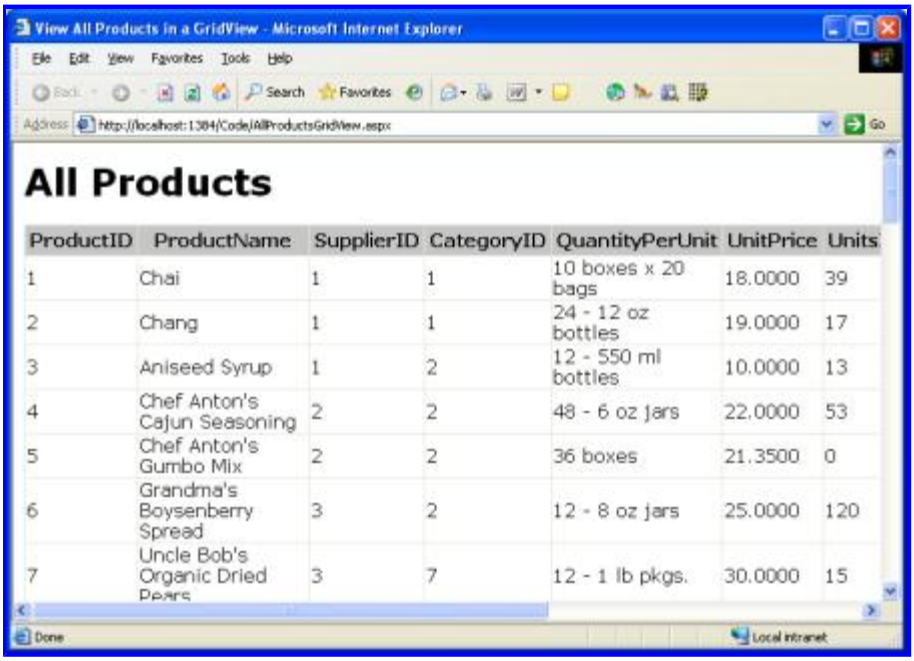

Figure 3: The List of Products is Displayed in a GridView

Step 3: Adding Field-Level Validation to the DataRow Classes

Field-level validation are checks that pertains to the property values of the business objects when inserting or updating. Some field-level validation rules for products include:

- The `ProductName` field must be 40 characters or less in length
- The `QuantityPerUnit` field must be 20 characters or less in length
- The `ProductID`, `ProductName`, and `Discontinued` fields are required, but all other fields are optional
- The `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` fields must be greater than or equal to zero

These rules can and should be expressed at the database level. The character limit on the `ProductName` and `QuantityPerUnit` fields are captured by the data types of those columns in the `Products` table (`nvarchar(40)` and `nvarchar(20)`, respectively). Whether fields are required and optional are expressed by if the database table column allows `NULL`s. Four check constraints exist that ensure that only values greater than or equal to zero can make it into the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, or `ReorderLevel` columns.

In addition to enforcing these rules at the database they should also be enforced at the DataSet level. In fact, the field length and whether a value is required or optional are already captured for each DataTable's set of DataColumns. To see the existing field-level validation automatically provided, go to the DataSet Designer, select a field from one of the DataTables and then go to the Properties window. As Figure 4 shows, the `QuantityPerUnit` DataColumn in the `ProductsDataTable` has a maximum length of 20 characters and does allow `NULL` values. If we attempt to set the `ProductsDataRow`'s `QuantityPerUnit` property to a string value longer than 20 characters an `ArgumentException` will be thrown.

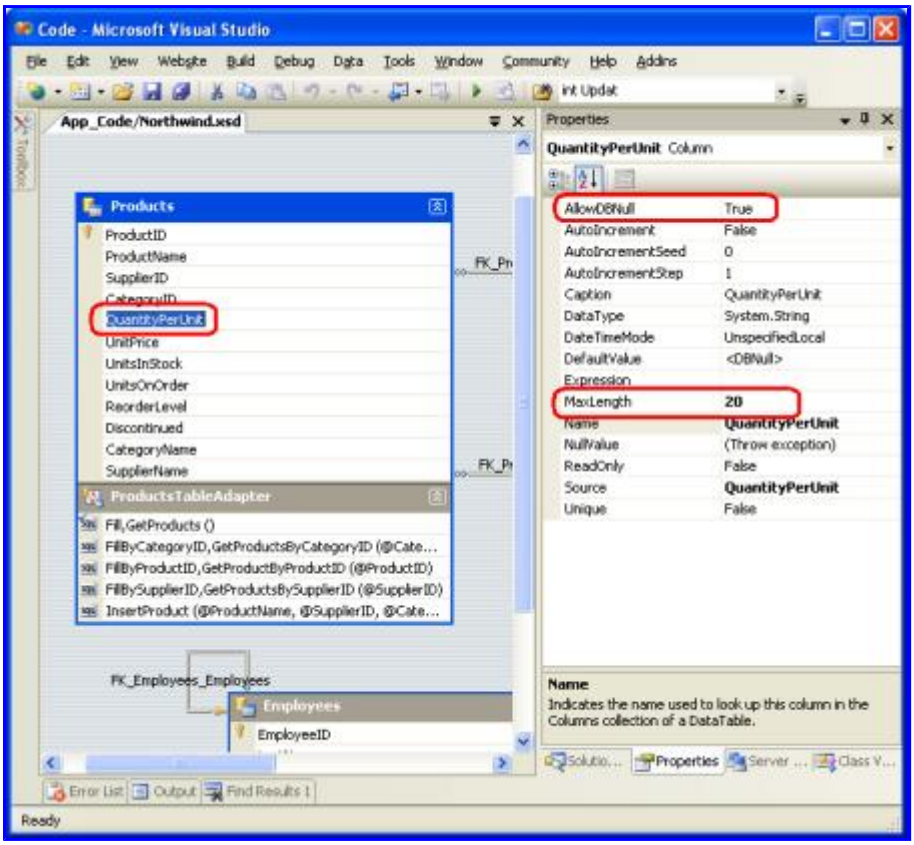

Figure 4: The DataColumn Provides Basic Field-Level Validation

Unfortunately, we can't specify bounds checks, such as the `UnitPrice` value must be greater than or equal to zero, through the Properties window. In order to provide this type of field-level validation we need to create an event handler for the DataTable's ColumnChanging event. As mentioned in the preceding tutorial, the DataSet, DataTables, and DataRow objects created by the Typed DataSet can be extended through the use of partial classes. Using this technique we can create a `ColumnChanging` event handler for the `ProductsDataTable` class. Start by creating a class in the `App_Code` folder named `ProductsDataTable.ColumnChanging.vb`.

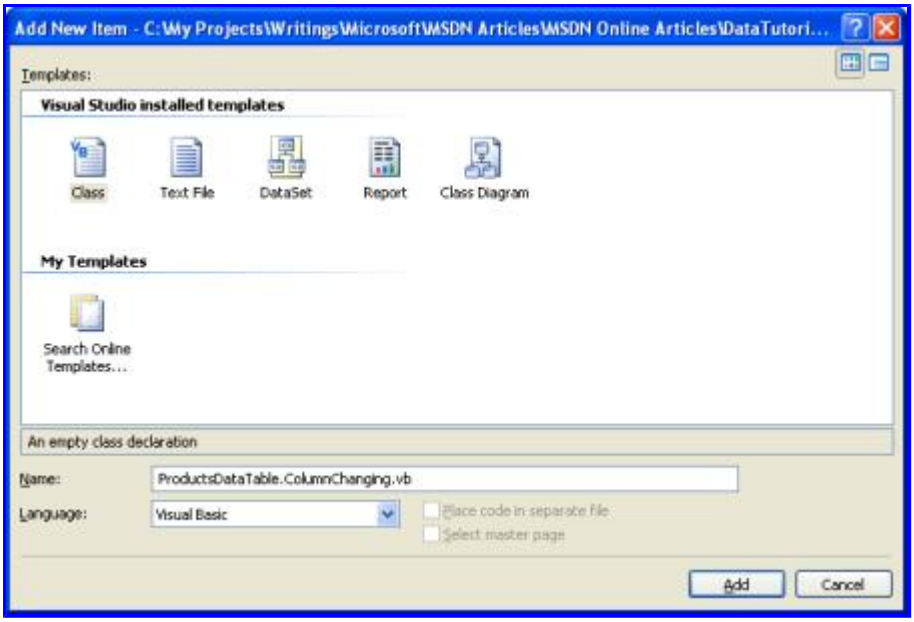

Figure 5: Add a New Class to the `App_Code` Folder

Next, create an event handler for the `ColumnChanging` event that ensures that the `UnitPrice`, `UnitsInStock`, `UnitsOnOrder`, and `ReorderLevel` column values (if not `NULL`) are greater than or equal to zero. If any such column is out of range, throw an `ArgumentException`.

ProductsDataTable.ColumnChanging.vb

```
Imports System.data

Partial Public Class Northwind
    Partial Public Class ProductsDataTable
        Public Overrides Sub BeginInit()
            AddHandler Me.ColumnChanging, AddressOf ValidateColumn
        End Sub

        Sub ValidateColumn(sender As Object, e As DataColumnChangeEventArgs)
            If e.Column.Equals(Me.UnitPriceColumn) Then
                If Not Convert.IsDBNull(e.ProposedValue) AndAlso _
                    CType(e.ProposedValue, Decimal) < 0 Then
                    Throw New ArgumentException( _
                        "UnitPrice cannot be less than zero", "UnitPrice")
                End If
            ElseIf e.Column.Equals(Me.UnitsInStockColumn) OrElse _
                e.Column.Equals(Me.UnitsOnOrderColumn) OrElse _
                e.Column.Equals(Me.ReorderLevelColumn) Then
                If Not Convert.IsDBNull(e.ProposedValue) AndAlso _
                    CType(e.ProposedValue, Short) < 0 Then
                    Throw New ArgumentException(String.Format( _
                        "{0} cannot be less than zero", e.Column.ColumnName), _
                        e.Column.ColumnName)
                End If
            End If
        End Sub
    End Class
End Class
```

Step 4: Adding Custom Business Rules to the BLL's

Classes

In addition to field-level validation, there may be high-level custom business rules that involve different entities or concepts not expressible at the single column level, such as:

- If a product is discontinued, its `UnitPrice` cannot be updated
- An employee's country of residence must be the same as their manager's country of residence
- A product cannot be discontinued if it is the only product provided by the supplier

The BLL classes should contain checks to ensure adherence to the application's business rules. These checks can be added directly to the methods to which they apply.

Imagine that our business rules dictate that a product could not be marked discontinued if it was the only product from a given supplier. That is, if product *X* was the only product we purchased from supplier *Y*, we could not mark *X* as discontinued; if, however, supplier *Y* supplied us with three products, *A*, *B*, and *C*, then we could mark any and all of these as discontinued. An odd business rule, but business rules and common sense aren't always aligned!

To enforce this business rule in the `UpdateProducts` method we'd start by checking if `Discontinued` was set to `True` and, if so, we'd call `GetProductsBySupplierID` to determine how many products we purchased from this product's supplier. If only one product is purchased from this supplier, we throw an `ApplicationException`.

```
<System.ComponentModel.DataObjectMethodAttribute_
    (System.ComponentModel.DataObjectMethodType.Update, True)> _
Public Function UpdateProduct( _
    productName As String, supplierID As Nullable(Of Integer), _
    categoryID As Nullable(Of Integer), quantityPerUnit As String, _
    unitPrice As Nullable(Of Decimal), unitsInStock As Nullable(Of Short), _
    unitsOnOrder As Nullable(Of Short), reorderLevel As Nullable(Of Short), _
    discontinued As Boolean, productID As Integer) _
    As Boolean

    Dim products As Northwind.ProductsDataTable = _
        Adapter.GetProductByProductID(productID)

    If products.Count = 0 Then
        Return False
    End If

    Dim product As Northwind.ProductsRow = products(0)

    If discontinued Then
        Dim productsBySupplier As Northwind.ProductsDataTable = _
            Adapter.GetProductsBySupplierID(product.SupplierID)

        If productsBySupplier.Count = 1 Then
            Throw New ApplicationException( _
                "You cannot mark a product as discontinued if it is " & _
                "the only product purchased from a supplier")
        End If
    End If

    product.ProductName = productName

    If Not supplierID.HasValue Then
        product.SetSupplierIDNull()
    Else
        product.SupplierID = supplierID.Value
```

```
    End If

    If Not categoryID.HasValue Then
        product.SetCategoryIDNull()
    Else
        product.CategoryID = categoryID.Value
    End If

    If quantityPerUnit Is Nothing Then
        product.SetQuantityPerUnitNull()
    Else
        product.QuantityPerUnit = quantityPerUnit
    End If

    If Not unitPrice.HasValue Then
        product.SetUnitPriceNull()
    Else
        product.UnitPrice = unitPrice.Value
    End If

    If Not unitsInStock.HasValue Then
        product.SetUnitsInStockNull()
    Else
        product.UnitsInStock = unitsInStock.Value
    End If

    If Not unitsOnOrder.HasValue Then
        product.SetUnitsOnOrderNull()
    Else
        product.UnitsOnOrder = unitsOnOrder.Value
    End If

    If Not reorderLevel.HasValue Then
        product.SetReorderLevelNull()
    Else
        product.ReorderLevel = reorderLevel.Value
    End If

    product.Discontinued = discontinued

    Dim rowsAffected As Integer = Adapter.Update(product)

    Return rowsAffected = 1
End Function
```

Responding to Validation Errors in the Presentation Tier

When calling the BLL from the presentation tier we can decide whether to attempt to handle any exceptions that might be raised or let them bubble up to ASP.NET (which will raise the `HttpApplication`'s `Error` event). To handle an exception when working with the BLL programmatically, we can use a Try...Catch block, as the following example shows:

```
Dim productLogic As New ProductsBLL()

Try
    productLogic.UpdateProduct("Scotts Tea", 1, 1, Nothing, _
      -14, 10, Nothing, Nothing, False, 1)
Catch ae As ArgumentException
    Response.Write("There was a problem: " & ae.Message)
```

```
End Try
```

As we'll see in future tutorials, handling exceptions that bubble up from the BLL when using a data Web control for inserting, updating, or deleting data can be handled directly in an event handler as opposed to having to wrap code in `Try...Catch` blocks.

Summary

A well architected application is crafted into distinct layers, each of which encapsulates a particular role. In the first tutorial of this article series we created a Data Access Layer using Typed DataSets; in this tutorial we built a Business Logic Layer as a series of classes in our application's `App_Code` folder that call down into our DAL. The BLL implements the field-level and business-level logic for our application. In addition to creating a separate BLL, as we did in this tutorial, another option is to extend the TableAdapters' methods through the use of partial classes. However, using this technique does not allow us to override existing methods nor does it separate our DAL and our BLL as cleanly as the approach we've taken in this article.

With the DAL and BLL complete, we're ready to start on our presentation layer. In the next tutorial we'll take a brief detour from data access topics and define a consistent page layout for use throughout the tutorials.

Happy Programming!

About the Author

Scott Mitchell, author of six ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer, recently completing his latest book, Sams Teach Yourself ASP.NET 2.0 in 24 Hours. He can be reached at mitchell@4guysfromrolla.com or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To…

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial include Liz Shulok, Dennis Patterson, Carlos Santos, and Hilton Giesenow. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.