

Strongly typed data revisited

William Vaughn

One of the ways developers can leverage the power of Visual Studio and the CLR is to create strongly typed data structures.

The last time I gave a session at the DevWeek conference I got a number of good questions on how to handle data validation and to improve overall data access performance. While my sessions discuss techniques to deal with these issues, I don't usually spend a lot of time on strongly typed data. Because of this continued interest, I decided to revisit this topic and make sure that developers have a better understanding of when this approach makes sense—and when it doesn't.

One of the problems developers are faced with is that while strongly typed data is discussed, in some cases the content is very dated—some published while Visual Studio .NET was still in “Alpha” stage. Much of the remaining content seems to be written in C# which makes it of limited usefulness for many Visual Basic .NET developers. This article provides an overview of the mechanics of strongly typed data management and how it can help developer productivity and application performance.

All applications need to set and return values from program variables. When you create a class in Visual Basic (or any CLR language) you describe the variable in detail.

```
Dim intYearsInService as Integer
```

When your code references this variable, the CLR uses the “address” of this variable and verifies that the data being moved there is conformant to the type and size of data defined for this variable. This ensures that the information stored is not going to overflow on to neighboring variables (or unused space), and that it conforms to the binary type defined for this memory address. Because you have defined the variable at design time, the compiler can “hard-code” the address and any conversions required.

When working with data structures such as DataSet, DataTable, and DataRow classes, unless you setup strongly typed data classes, the compiler needs to take extra time at runtime to determine where to place incoming data and how to convert it to the destination data type. For example, if you extract a value from a DataRow using the following syntax, ADO.NET must take an order of magnitude more cycles to resolve the address and fetch the data (when compared to a strongly typed reference).

```
txtYears.Text = myDataSet.Tables("Info").Rows(0).Item("YearsInService")
```

Using a strongly typed DataSet class to refer to the same data would be coded as follows:

```
txtYears.Text = mySTData.Info(0).YearsInService
```

The result is far less code being executed behind the scenes to resolve the address and better human-readability at the same time. During coding, developers working with strongly typed data are also assisted with copious support by intellisense. This is not the case with “untyped” data structures (the default).

To implement strongly typed DataSets, you need to create a class in your application that declares the DataSet, the DataRow and DataColumn structures in memory. Yes, you could code this yourself, but there's no need to because it's easy to get Visual Studio .NET to do it for you. Of course, if you're paid by the line of code you write—go for it (your mileage may vary).

The following steps walk you through one of the numerous ways to create a strongly typed DataSet against a single data table. There are also other ways to create strongly typed DataSets—most of which involve similar approaches using Visual Studio .NET and drag-and-drop. Yes, returning all the rows from any table causes a number of other issues that I don't need to belabor here. However, setting up parameter queries with the DACW can cause additional problems that we don't have the space here to address—perhaps next time.

Using the Server Explorer

The simplest first step when creating a strongly typed DataSet is to simply drag a data table icon from the Server Explorer to an open Form in Visual Studio. This automatically adds code to your application Form to instantiate a data connection (depending on the connection you chose in the Server Explorer) and a fully configured DataAdapter. The downside to this approach is that Visual Studio .NET also generates a SELECT statement that returns *all* rows and columns from the table—whether the target data table has 60 rows or 60,000,000 rows. This approach is fine for smaller (< 1000 rows) tables.

No, dragging a stored procedure from the server explorer does not generate a DataAdapter which is needed to help Visual Studio .NET generate a strongly typed DataSet.

Once you've created the DataAdapter, click on the form and the newly-created DataAdapter, and choose "Generate DataSet...". Jump down to "Creating a Strongly Typed DataSet" (later in this article) for details and the next steps you need to take to create a strongly typed DataSet.

Using the DataAdapter Configuration Wizard

In situations where you want more control over what rows and columns are selected or when you want to use a parameter query to fetch a more focused rowset, you'll want to use the Visual Studio .NET's Data Access Configuration Wizard (DACW) to generate the DataAdapter as a first step when creating strongly typed data structures.

Before you get started, it's a good idea to setup a connection to your target server using a suitable data provider. Depending on the type of .NET Data Provider you choose, Visual Studio .NET might not be able to run the DACW as not all providers support the additional functionality needed to construct the data class.

Start by dragging your .NET Data Provider's implementation of the DataAdapter to the form from the Toolbox Data pane. For example, for the SqlClient .NET Data Provider, drag the SqlDataAdapter to the form. This launches the DACW.

Next, follow the dialogs to choose the connection you setup earlier.

Next, for purposes of this demo we'll use SQL statements to define the SelectCommand generated by the DACW. Yes, you can also use existing stored procedures or get Visual Studio .NET to generate appropriate stored procedures to fetch the data. Click Next.

Now it's time to enter the SQL needed to return data from the database table. Use the dialog to hand-code the SELECT statement or click on the Query Builder to let Visual Studio .NET assist in the creation of this query. No, don't refer to more than one table or try to JOIN to another table. Yes, it's possible to limit the scope of the query (which is a great idea), and that can be done by coding a WHERE clause that contains one or more input parameters that must be set at runtime prior to executing the DataAdapter Fill method. The Advanced Options... button in this dialog opens a dialog that lets you specify whether or not Visual Studio .NET should add additional code to create the action queries needed to change the data, manage concurrency and refresh the DataSet. Again, leave these settings in their default state to keep things simple.

When you click "Next", the DACW generates the code needed to construct the DataAdapter and all of its associated SELECT and action commands. We're almost done.

Creating a Strongly Typed DataSet

Once you have created the new DataAdapter and set focus to the Form, you can click on the new DataAdapter in the tool tray and click on "Generate DataSet" or choose Data | Generate DataSet... from the menu. This instructs Visual Studio .NET to build the class library that represents the DataTable, its associated DataRow and DataColumn objects along with appropriate properties, methods and events to support the DataSet—inherited from the DataSet base class. Visual Studio .NET also constructs an XSD file that represents the strongly typed DataSet in XML.

The "Generate DataSet..." dialog asks whether you want to redefine an existing DataSet or create a new one. Choose "New:" and enter a name. I usually begin my DataSet names with "ds"—like dsCustomers. Choose the DataAdapter DataTable(s) to be included in your new DataSet class and click OK.

From this point forward you can begin seeing the benefits from the generated code—all of it written by Visual Studio .NET. No, you don't have to tell your boss that all of this code was generated by the DACW. That's between you and your conscience.

Addressing a Strongly Typed DataSet

Once your Visual Studio .NET generated DataSet class is in place you can instantiate a variable to point to it and let the compiler take it from there. Here's a code example that I used to instantiate and reference the newly created DataSet. I used drag-and-drop (D&D) to generate the SqlDataAdapter against the "ValidStates" table. This table contains about 65 rows—one for each valid "State Code" as defined by the US Postal Service. It's a great candidate for the D&D approach.

```
Dim stDS As New dsValidStates()  
  
Private Sub Form1_Load(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles MyBase.Load  
    SqlDataAdapter1.Fill(stDS) ' Run the query and populate the DataTable in the
```

```
        ' Strongly typed DataSet
    txtStateCode.Text = stDS.ValidStates(0).StateCode
    txtStateName.Text = stDS.ValidStates(0).State
End Sub
```

Listing 1: Coding a strongly typed DataSet.

As you can see from this code, it's not that hard to instantiate the auto-generated DataSet class. When the "Generate DataSet..." dialog appeared, I named the generated DataSet "dsValidStates". The first Dim statement declares an instance of that DataSet. After the code uses the Fill method, ADO.NET populates the DataTable associated with the *dsValidStates* instance (named stDS). Next, the code references two columns in the DataSet and moves the data to TextBox controls.

This code is more human-readable and considerably faster to execute than its untyped equivalent. That's the advantage of strongly typed DataSets. The disadvantage is that once the class is generated (at design time), it assumes that the inbound data schema will not change sometime in the future. If your data structures are still in flux, it might seem pretty tedious to use this approach to generate code to return data. In addition, while this approach can be used to generate more complex DataSets, as your SELECT queries get more complex, you start running up against the DataAdapter stops. That is, if your SelectCommand query contains a DISTINCT clause, JOIN clause or other syntax that makes it unclear as to where the data is sourced, the DACW can give up trying to generate the DataAdapter or the action commands used to update the data.

This article focused on the basics of creating and referencing strongly typed DataSets. This approach can help make your coding easier—easier to create, read and easier for the CLR to execute efficiently.