Pro ADO.NET 2.0

Sahil Malik

Apress®

Pro ADO.NET 2.0

Copyright © 2005 by Sahil Malik

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN: 1-59059-512-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Jonathan Hassell
Technical Reviewers: Frans Bouma and Erick Sgarbi
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Emily K. Wolman
Copy Edit Manager: Nicole LeClerc
Copy Editor: Linda Marousek
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Kinetic Publishing Services, LLC
Proofreader: April Eddy
Indexer: Carol Burbo
Artist: Kinetic Publishing Services, LLC
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 3

ADO.NET Hello World!

The first chapter explained the purpose of ADO.NET and where it fits in the overall architecture. It explained using common block diagrams, the very high-level structure of ADO.NET, the connected and disconnected portions of ADO.NET, and the .NET data provider model.

The second chapter took that discussion from a 30,000-ft. view to about a 10,000-ft. view where you saw the various objects, their inheritance layout within ADO.NET, the various namespaces, and the reasoning behind that structure.

It's now time to walk on the ground and write a few real data-driven applications. But before you deal with the intricacies and complexities of a true enterprise-level data-driven architecture, it makes sense to see a few simple applications first. This is in the spirit of "crawl before you walk, walk before you run." This chapter begins with extremely simple data-driven applications that require you to write absolutely no code at all. In fact, an entire working application is produced by only dragging and dropping. Then, this approach is taken forward to increasingly more involved examples where you'll blend some writing with dragging and dropping. You'll proceed to a small data-driven console application where you'll write every part of the code yourself.

Since all the examples presented in this chapter will be data driven, it is probably a good idea to set up the data source being used first. The examples presented in this book exemplify various ADO.NET concepts using a local Microsoft SQL Server 2005 instance. Any differences with Oracle or other major providers will be identified as they arise. For the purposes of this chapter, however, the examples are simple enough that there are no differences between the major data providers.

Setting Up the Hello World Data Source

The quintessential Hello World example serves as a simple introduction to any programming concept. Let's try and leverage that to our advantage by setting up a simplistic data source. As mentioned earlier, the data source will be a database on the local running instance of Microsoft SQL Server 2005. The database name will be `Test`, and it will contain one table called `Demo`. This table can be set up using the following script:

```
Create Table Demo
(
DemoID int identity primary key,
DemoValue varchar(200)
)
```

```
Go

Insert Into Demo (DemoValue) Values ('Hello World')
GO
```

The Demo table contains two columns, one of which is DemoID of int type, which is an identity column. Identity columns are specific to SQL Server; if you're working with Oracle, you'll have to modify your queries to use sequences instead. Thus, depending on the exact database you're working with, you'll have to leverage either an identity or a sequence. For instance, in IBM DB2 you have a choice of picking between a sequence and an identity.

The second column is DemoValue of VarChar(200) type, which stores a simple text value. You can download the previous script from the code samples for this chapter in the Downloads section of the Apress website (http://www.apress.com); it can be found under Chapter 3 in a file called Create Database.sql. This will enable you to set up such a data source for a local running instance of Microsoft SQL Server 2005.

Finally, there's one row inserted in the table—the Hello World row. You can run a simple SELECT query and find the contents of the underlying data source of your Hello World applications that you'll be writing in this chapter. Figure 3-1 shows the contents of the underlying data source.

Figure 3-1. *Contents of the underlying data source for the examples in this chapter*

With the data source set up, let's begin by creating the first simplistic data-driven application.

Creating a Data-Driven Application: The Drag-and-Drop Approach

The purpose of this application is simple. All it needs to do is provide you with a user interface that lets you view the contents of the Demo table in the Test database. Also, it should give you some user interface that lets you modify the data.

Historically, developed applications have taken two diverse paths. One insists on being a monolithic, fat-client architecture that leverages the power of the desktop. Obviously, the advantage here is the flexibility you get by having the full power of the desktop and the local nature of the application. The disadvantages are deployment and maintenance issues.

The second kind of application is designed to work on various platforms through a browser. Typically, these applications are HTML-based, which leverage very little power of the end client, and most of the work is done at the server. The advantages are easy deployment and versioning, but the disadvantages include a colossal waste of the end client's computing power, and your application having to support various configurations at the end client that you cannot control.

In addition, there are some midway application architectures, such as ActiveX and ClickOnce. But for the purposes of this chapter, let's concentrate on the two major kinds of applications: web-based (ASP.NET) and Windows Forms–based.

Let's begin by looking at web-based applications first, or as they are referred to in the .NET platform, ASP.NET applications.

Drag and Drop in ASP.NET 2.0

The simplicity of creating an application using drag and drop in ASP.NET 2.0 is quite remarkable. The approach and underlying architecture are different from ASP.NET 1.1, which will not be discussed here.

The code for this example can be downloaded from the Downloads section for this chapter under `DragDropWebsite`, or you can easily create it yourself by following the steps here. Do note that because there's no code to write, the instructions are exactly the same for both C# and VB.NET:

1. Start by firing up Visual Studio 2005 and creating a new website. To do this, select File ➤ New ➤ WebSite.

2. Give it a name; I chose `DragDropWebsite`, which is demonstrated in Figure 3-2.

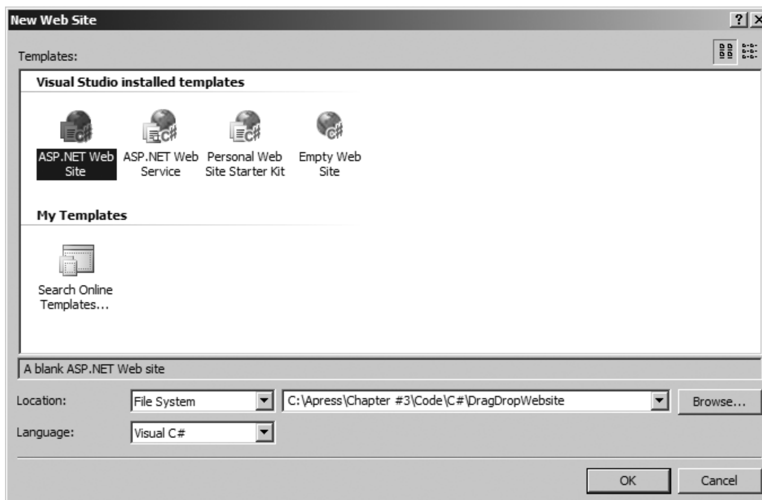

Figure 3-2. *Creating a new ASP.NET 2.0 website*

Take note that I am using C# in Figure 3-2 as my choice language. Since the steps are exactly the same for VB.NET, if you prefer, you can choose VB.NET as your choice language.

3. Click OK to create the website.

4. The created website should have one `Default.aspx` page created for you. If that page is not already open inside the Visual Studio IDE, double-click it to open it. At the bottom of that page, click Design to view a blank page.

5. Next, open the Database Explorer/Server Explorer window. If the window is not already visible, you can enable it by selecting View ➤ Server Explorer. This window should look similar to the one shown in Figure 3-3.

Figure 3-3. *The Database Explorer/Server Explorer window*

If your Database Explorer/Server Explorer window looks somewhat different than mine—don't panic. It's only because your Server Explorer already has a few entries and mine is blank.

6. Next, right-click on Data Connections and choose Add Connection as shown in Figure 3-4.

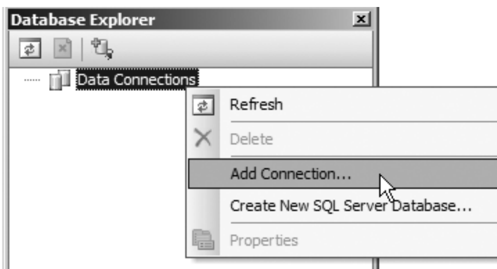

Figure 3-4. *Adding a connection to the Database Explorer/Server Explorer*

This should show a dialog box that prompts you to select the appropriate data source. At this point, you can go ahead and fill in the various values. When you click Test Connection, you should see a dialog box informing you that the connection is valid and the test succeeded, as shown in Figure 3-5.

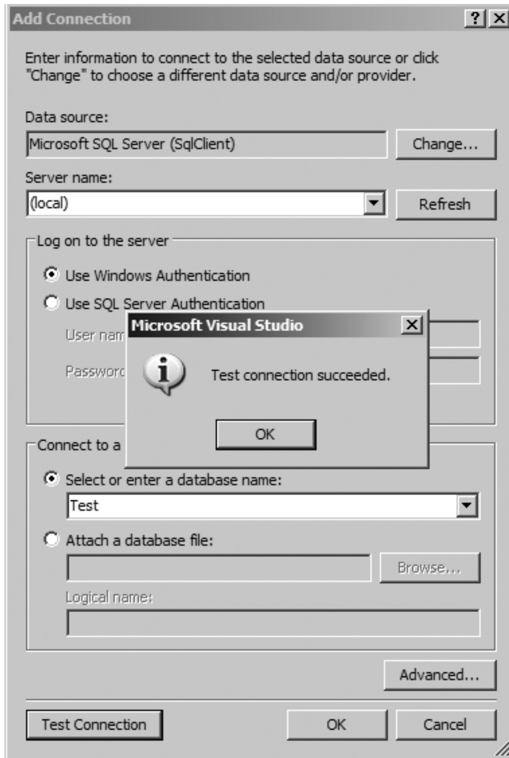

Figure 3-5. *Setting up the data source*

7. Click OK twice to accept the data source. At this point, you should be able to see a data source defined under the Data Connections node in the Database Explorer window.

8. Next, expand the Database Explorer's newly added data source until you see the `Demo` table. This can be seen in Figure 3-6.

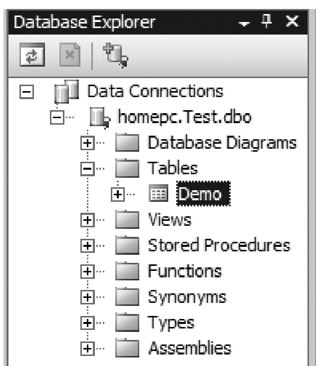

Figure 3-6. *The Demo table—setting up the data source*

9. Now the fun starts! With your mouse, drag and drop the `Demo` table to the surface of `Default.aspx`. This should add two things on the surface of the `.aspx` page: a `GridView` control and a `SqlDataSource`. This can be seen in Figure 3-7.

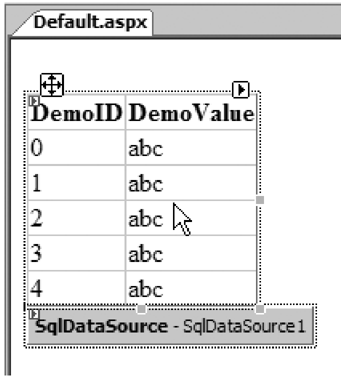

Figure 3-7. *The newly added GridView and SqlDataSource controls on the .aspx page*

Notice the small, black, arrow-like button facing to the right at the top-right edge of the `GridView` control. When you click it, you should see an editing pane that allows you to format the `GridView` control and set a few properties. Go ahead and enable editing and deleting, along with formatting the `GridView` to a scheme of your choice. This can be seen in Figure 3-8.

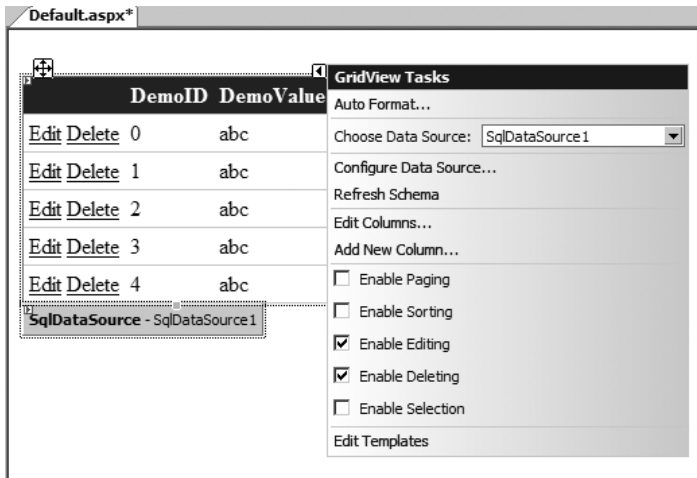

Figure 3-8. *Setting properties on the GridView*

10. That's it. Now compile and run the application. You might be prompted to start with or without debugging, just select the default choice. You should see your Hello World web-based application running, as shown in Figure 3-9.

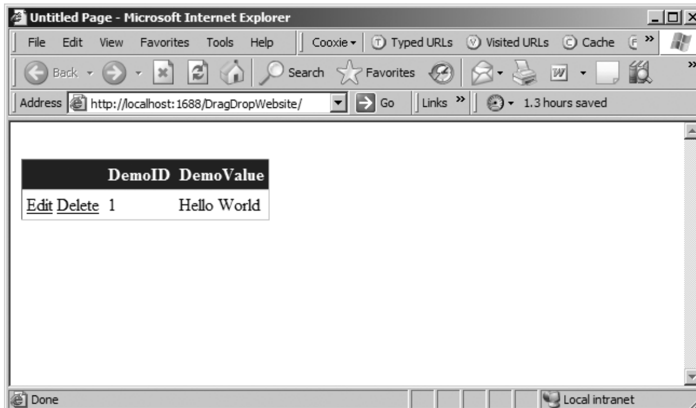

Figure 3-9. *Hello World in ASP.NET*

Try playing around with the application a bit. You'll notice that this is a fully functional application that even lets you modify the underlying data. If you do happen to modify the underlying data, you can easily restore it by running the script provided with this chapter again.

So you wrote absolutely no C# or VB.NET code and you have a data-driven web-based application ready. How did it all work? Well, the framework did all the work for you. It queried the underlying data source for you and encapsulated all that functionality within the various properties set on the `GridView` control and the `SqlDataSource` control.

If you view the source of the `.aspx` page and check out the listing for `SqlDataSource`, you'll see that it looks like the code shown in Listing 3-1.

Listing 3-1. *The SqlDataSource Control Defined on the Default.aspx Page*

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%$ ConnectionStrings:TestConnectionString1 %>"
    DeleteCommand="DELETE FROM [Demo] WHERE [DemoID] = @original_DemoID"
    InsertCommand="INSERT INTO [Demo] ([DemoValue]) VALUES (@DemoValue)"
    ProviderName="<%$ ConnectionStrings:TestConnectionString1.ProviderName %>"
    SelectCommand="SELECT [DemoID], [DemoValue] FROM [Demo]"
    UpdateCommand=
      "UPDATE [Demo] SET [DemoValue] =
      @DemoValue WHERE [DemoID] = @original_DemoID">
    <InsertParameters>
       <asp:Parameter Name="DemoValue" Type="String" />
    </InsertParameters>
    <UpdateParameters>
       <asp:Parameter Name="DemoValue" Type="String" />
       <asp:Parameter Name="original_DemoID" Type="Int32" />
    </UpdateParameters>
    <DeleteParameters>
       <asp:Parameter Name="original_DemoID" Type="Int32" />
    </DeleteParameters>
 </asp:SqlDataSource>
```

A few things are worth noting in Listing 3-1:

- The *connection string*, which is the information that tells the underlying libraries what data source to connect with and how to connect with it, has already been set up for you under `ConnectionStrings:TestConnectionString1`. This, as it turns out, has been specified in the `Web.Config` file under the `ConnectionStrings` section for you. This can be seen in Listing 3-2.

Listing 3-2. *The Connection String Defined for You in the Web.Config File*

```
<connectionStrings>
   <add name="TestConnectionString1"
   connectionString="Data Source=(local);Initial Catalog=Test;
     Integrated Security=True" providerName="System.Data.SqlClient"/>
</connectionStrings>
```

- The framework queried the underlying data source for you and prepared SQL statements for the various possible commands. This can be seen in Listing 3-1.
- Those commands are even parameterized with the right data types. All of this—written for you, by the framework. This also can be seen in Listing 3-1.

Next, let's turn our attention to the `GridView` control added for you by the framework. If you look in the source of the `.aspx` page, you should see code similar to that in Listing 3-3.

Listing 3-3. *The GridView Control Defined on the Default.aspx Page*

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
BackColor="White" BorderColor="#CCCCCC" BorderStyle="None" BorderWidth="1px"
CellPadding="4" DataKeyNames="DemoID" DataSourceID="SqlDataSource1"
EmptyDataText="There are no data records to display." ForeColor="Black"
GridLines="Horizontal">
   <FooterStyle BackColor="#CCCC99" ForeColor="Black" />
   <Columns>
      <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
      <asp:BoundField DataField="DemoID" HeaderText="DemoID"
         ReadOnly="True" SortExpression="DemoID" />
      <asp:BoundField DataField="DemoValue"
         HeaderText="DemoValue" SortExpression="DemoValue" />
   </Columns>
   <PagerStyle BackColor="White" ForeColor="Black" HorizontalAlign="Right" />
   <SelectedRowStyle BackColor="#CC3333" Font-Bold="True" ForeColor="White" />
   <HeaderStyle BackColor="#333333" Font-Bold="True" ForeColor="White" />
</asp:GridView>
```

As you can see from Listing 3-3, the data source for the `GridView` control has been defined as the `SqlDataSource1` object, which is what you see in Listing 3-1. This is what binds the `GridView` and the data source together. Then it's just a question of adding the relevant bound columns and the command buttons and your application is ready to run!

Thus, by a simple drag-and-drop operation, you're able to create a data-driven application from the ground up with very little code.

Caution You just created a data-driven application. Why should you bother to read any further? At this point, I must goad you to continue. A little knowledge is a dangerous thing, and you should not leave your ADO.NET knowledge incomplete because you now have the power to create a data-driven application in a matter of minutes by a simple point-and-click operation. As you'll learn in future chapters, your application is just as good as the amount of effort you put into it. You can't expect drag-and-drop applications to help you create a well-architected enterprise-level application up and running; however, it's important to learn this approach and possibly leverage it to implement a fast track to your eventual goal.

Now, with the data-driven ASP.NET application set up, let's look at a Windows Forms–based application created in a similar fashion.

Drag and Drop in a Windows Forms Application

Similar to an ASP.NET data-driven application, let's go ahead and follow a few simple steps to create a data-driven Windows Forms application. You can download the necessary code for this application from the associated code download in `DragDropWinApp`; however, since it doesn't make much sense to look at the final code, because it is all autogenerated anyway, I recommend that you follow these steps:

1. Begin by creating a new Windows Forms application in the language of your choice. Call it `DragDropWinApp`. Open the `Form1` form added for you in Design mode.

2. Next, within Visual Studio 2005, go to the Data Sources window. If this window is not visible by default, then select Data ➤ Show Data Sources, as shown in Figure 3-10.

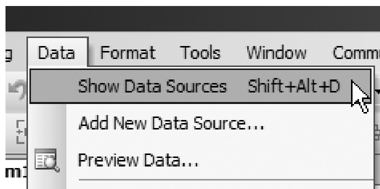

Figure 3-10. *The Show Data Sources menu item*

When you do see this window, as shown in Figure 3-11, click the Add New Data Source link.

Figure 3-11. *The Data Sources window*

3. When prompted to choose the Data Source type, select Database, as shown in Figure 3-12.

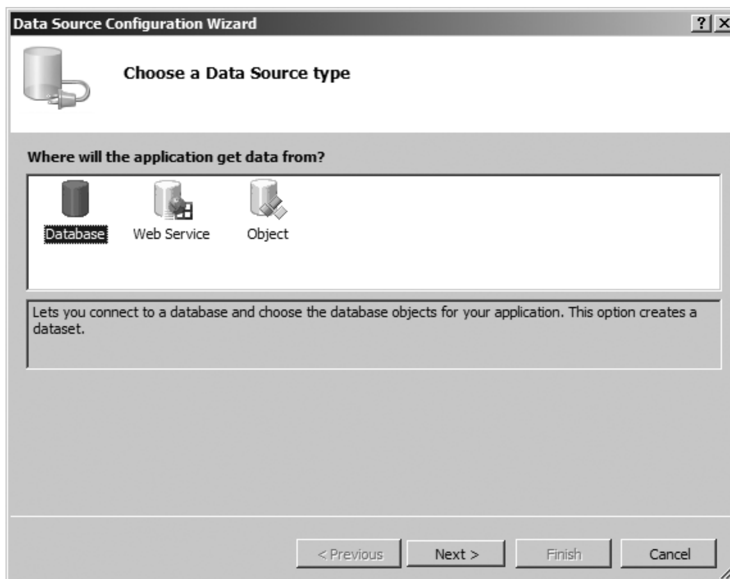

Figure 3-12. *Choosing the Data Source type*

4. When prompted to choose your data connection (see Figure 3-13), either choose the connection if it is already available in the list or click New Connection to add a new connection in a dialog box very much like Figure 3-5, which you saw in the ASP.NET drag-and-drop application.

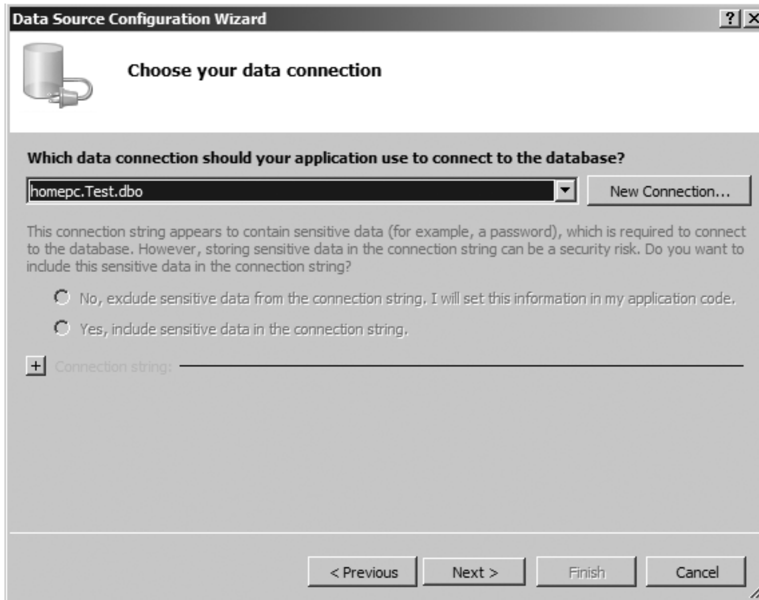

Figure 3-13. *Choosing the data connection*

5. When prompted, choose to save the connection string, as shown in Figure 3-14.

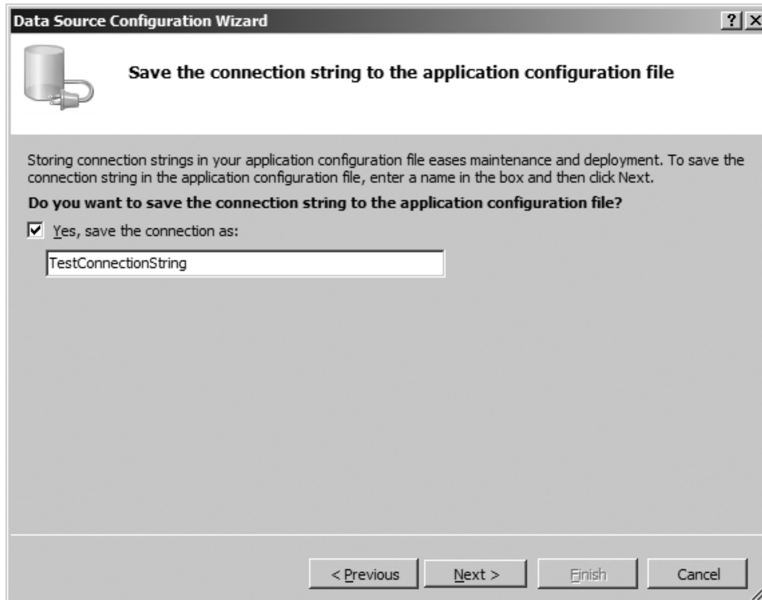

Figure 3-14. *Choosing to save the connection string*

6. When prompted to choose the database objects in your data source, choose the Demo table, as shown in Figure 3-15.

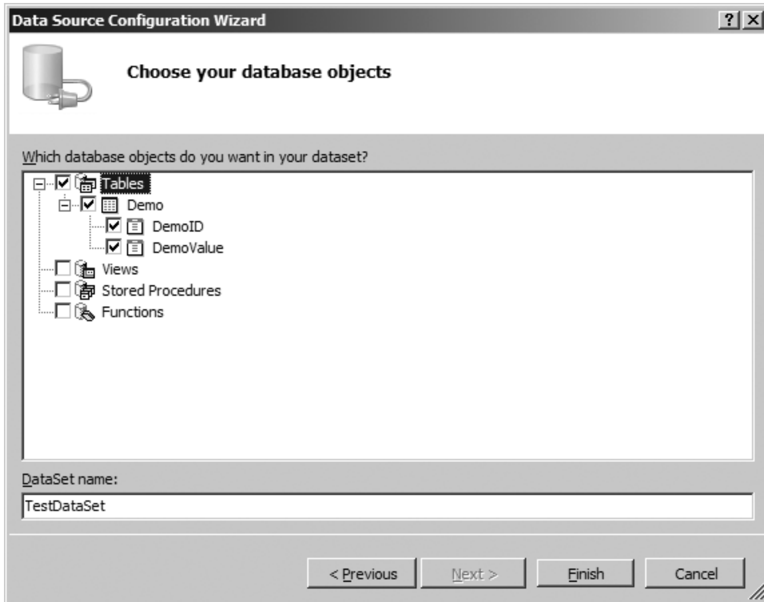

Figure 3-15. *Choosing the Demo table to be a part of your data source*

7. Click Next and Finish, which adds the `TestDataSet` data source to your application. At this point, you should see the `TestDataSet` data source added in the Data Sources window. If you select the `Demo` table under the data source, you can see a drop-down arrow next to it. For the purposes of this application, you can select `DataGridView`, as shown in Figure 3-16.

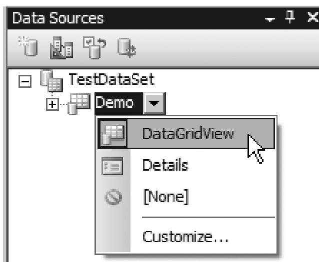

Figure 3-16. *Configuring the data source's Demo table*

8. Next, drag and drop the `Demo` table onto the surface of the form. This operation should add a number of controls to the surface of the form, as shown in Figure 3-17 (after rearranging them a bit).

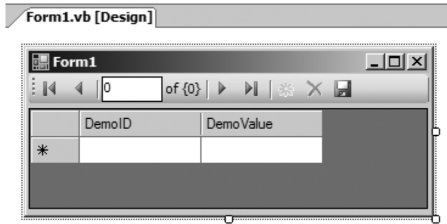

Figure 3-17. *The form in Design mode after rearranging the autogenerated controls*

This operation also adds a number of controls in the component tray under the form, as shown in Figure 3-18.

Figure 3-18. *Various controls added for you in the component tray*

9. That's it. Your data-driven application is ready. Compile and run it to see a fully operational window, as shown in Figure 3-19. You'll also see that you can edit the underlying data using this application. As an exercise to the reader, you can repeat this application by selecting something other than `DataGridView` in step 7.

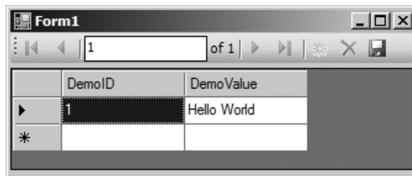

Figure 3-19. *A fully running data-driven Windows Forms application, created using drag and drop*

Again, you just created a fully functional data-driven application without actually having to write any code. As it turns out, in this case, the framework actually wrote some code for you. If you open the `App.Config` file, you'll see that the application has saved the connection string as an element, as shown in Listing 3-4.

Listing 3-4. *The Connection String in the App.Config File*

```
<connectionStrings>
     <add name="DragDropWinApp.Settings.TestConnectionString"
      connectionString=
      "Data Source=(local);Initial Catalog=Test;Integrated Security=True"
         providerName="System.Data.SqlClient" />
 </connectionStrings>
```

Also, if you view the added form's code, you'll see the code as shown in Listings 3-5 and 3-6.

Listing 3-5. *Autogenerated Code in C#*

```
private void bindingNavigatorSaveItem_Click(object sender, EventArgs e)
{
   if (this.Validate())
   {
      this.demoBindingSource.EndEdit();
      this.demoTableAdapter.Update(this.testDataSet.Demo);
   }
   else
   {
      System.Windows.Forms.MessageBox.Show(this, "Validation errors occurred.",
      "Save", System.Windows.Forms.MessageBoxButtons.OK,
      System.Windows.Forms.MessageBoxIcon.Warning);
   }

}

private void Form1_Load(object sender, EventArgs e)
{
   // TODO: This line of code loads data into the 'testDataSet.Demo' table.
   // You can move, or remove it, as needed.
   this.demoTableAdapter.Fill(this.testDataSet.Demo);

}
```

Listing 3-6. *Autogenerated Code in Visual Basic .NET*

```
Private Sub bindingNavigatorSaveItem_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles bindingNavigatorSaveItem.Click
   If Me.Validate Then
      Me.DemoBindingSource.EndEdit()
      Me.DemoTableAdapter.Update(Me.TestDataSet.Demo)
   Else
      System.Windows.Forms.MessageBox.Show(Me, "Validation errors occurred.", _
      "Save", System.Windows.Forms.MessageBoxButtons.OK, _
      System.Windows.Forms.MessageBoxIcon.Warning)
   End If

End Sub

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) _
   Handles MyBase.Load
   'TODO: This line of code loads data into the 'TestDataSet.Demo' table.
   ' You can move, or remove it, as needed.
```

```
    Me.DemoTableAdapter.Fill(Me.TestDataSet.Demo)

End Sub
```

This is not the only code generated for you. As you'll see in subsequent chapters, there is a lot of code generated in the `TestDataSet` strongly typed `DataSet` and in a hidden file called `Form1.Designer.cs` or `Form1.Designer.vb`. But for now, let's leave that for later.

Next, let's look at an application that gets a little bit more hands on as far as writing code yourself goes.

Hybrid Approach: Write Some Code, Do Some Drag and Drop

In the previous example, you saw how to easily create a data-driven Windows Forms application by simply dragging and dropping the various components onto the surface of the form. The important part to realize here is that the code that is autogenerated for you is "one size fits all." It's a lot of very generic code that is written in such a way that it will work in a logically correct manner for most applications; however, it might not be the most efficient code. Of course, when it works for most situations that implies that there will be that one odd situation where it won't work. Can you imagine your enterprise application having 500 tables? And then imagine having to drag and drop those 500 tables in a drag-and-drop architecture? It's just not maintainable. Plus, you can't customize that code to fit any situation that you may be faced with.

However, this approach does have its place in application architecture. Depending on your situation, you may decide that creating a full-fledged application using only a drag-and-drop application is a bad idea, but you could leverage the autogenerated code to your advantage by using the various generated objects as shortcuts and writing code similar to what has been shown in Listings 3-5 and 3-6 yourself.

Let's look at an example that demonstrates this very hybrid approach. You can download this application from the associated code download in `ConsoleApp`, or you can follow these steps to create such an application yourself:

1. Begin by creating a new console application. Call it `ConsoleApp`.
2. Follow steps 2 through 7 of the Windows Forms application to add a new data source. Since, in this exercise, you'll write code in a hybrid approach (drag-and-drop plus write code) as a part of a console application, it doesn't make sense to set the `Demo` table to `DataGridView` or anything else.
3. In the console application, write code as shown in Listings 3-7 and 3-8. This code will be used to fill in the contents of the `Demo` table into the `testDS.Demo` object, and then write out the value of the first row's `DemoValue` column.

Listing 3-7. *Code to Fill and Write Hello World from the Data Source in C#*

```
TestDataSet testDS = new TestDataSet();
TestDataSetTableAdapters.DemoTableAdapter tableAdapter =
   new TestDataSetTableAdapters.DemoTableAdapter();
tableAdapter.Fill(testDS.Demo);
```

```
TestDataSet.DemoRow demoRow =
   (TestDataSet.DemoRow)testDS.Demo.Rows[0];
Console.WriteLine(demoRow.DemoValue);
```

Listing 3-8. *Code to Fill and Write Hello World from the Data Source in Visual Basic .NET*

```
Dim testDS As TestDataSet = New TestDataSet()
Dim tableAdapter As TestDataSetTableAdapters.DemoTableAdapter = _
      New TestDataSetTableAdapters.DemoTableAdapter()
tableAdapter.Fill(testDS.Demo)

Dim demoRow As TestDataSet.DemoRow = _
   CType(testDS.Demo.Rows(0), TestDataSet.DemoRow)
Console.WriteLine(demoRow.DemoValue)
```

4. Compile and run the application. You should see an output as shown in Figure 3-20.

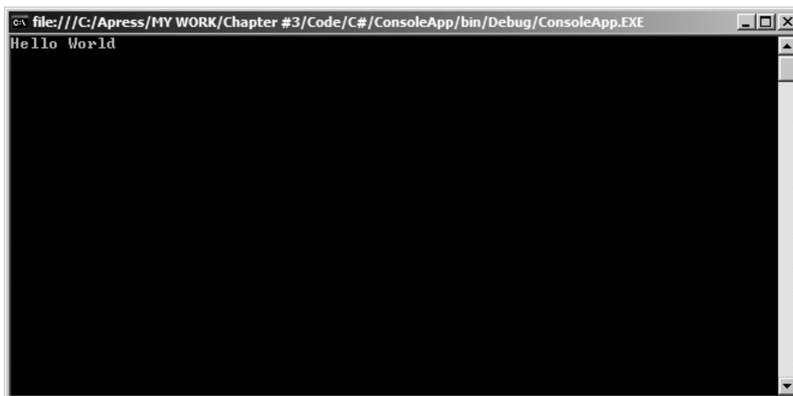

Figure 3-20. *The running hybrid application*

Thus, as you can see, you were able to write a few lines of code to leverage a lot of auto-generated code. Just as an exercise, read the code closely and try and understand what it does. There are really only four steps involved:

1. The first is to create an instance of `TestDataSet`:

C#
```
TestDataSet testDS = new TestDataSet();
```

VB.NET
```
Dim testDS As TestDataSet = New TestDataSet()
```

2. The second step is to create a new instance of the autogenerated `DemoTableAdapter` object. Note that this is an autogenerated object, which means this is a table adapter that is specific to your situation and the table you specified. As you'll see in Chapter 9, the framework actually wrote a lot of code for you to make this possible. In other words, this code is not a native part of the .NET Framework. Instead, it builds upon existing .NET Framework classes to provide you with a class, the `DemoTableAdapter` class, that is specific to your purpose and situation:

   C#

   ```
   TestDataSetTableAdapters.DemoTableAdapter tableAdapter =
      new TestDataSetTableAdapters.DemoTableAdapter();
   ```

   VB.NET

   ```
   Dim tableAdapter As TestDataSetTableAdapters.DemoTableAdapter = _
         New TestDataSetTableAdapters.DemoTableAdapter()
   ```

3. The third step is to use the `DemoTableAdapter` to fill in the `testDS.Demo` table. As you will see in Chapters 6 and 7, this fill operation is really being done by an underlying object called the `DataAdapter`, but in this case, the framework masks all these complexities from you. Obviously, if you needed deeper-level control (for instance, working with hierarchical data or other such situations), then this approach won't work for you. Chapter 10 covers an instance using hierarchical data where you could not possibly use this approach correctly enough to work with a relatively more complex data structure:

   C#
   ```
   tableAdapter.Fill(testDS.Demo);
   ```

   VB.NET
   ```
   tableAdapter.Fill(testDS.Demo)
   ```

4. And the final step is to query the filled object's first row's `DemoValue` column. This is very much like querying a `DataSet`. This is because `TestDataSet` is really nothing but a class that inherits from `DataSet`. It is also referred to as a strongly typed `DataSet`, which is covered in depth in Chapter 6:

   C#

   ```
   TestDataSet.DemoRow demoRow =
      (TestDataSet.DemoRow)testDS.Demo.Rows[0];
   Console.WriteLine(demoRow.DemoValue);
   ```

   VB.NET

   ```
   Dim demoRow As TestDataSet.DemoRow = _
      CType(testDS.Demo.Rows(0), TestDataSet.DemoRow)
   Console.WriteLine(demoRow.DemoValue)
   ```

This code gives you a little more flexibility than a pure drag-and-drop approach. But you still can't appreciate what is going on behind the scenes without actually diving deeper into the depths of ADO.NET.

For instance, *where is the SQL Query in the previous code*?

As you'll see in Chapter 9, the SQL Query is embedded deep inside the autogenerated code for `TestDataSet`. But let's leave that for Chapter 6. For now, let's look at a simple but purely write-yourself approach and create a simple application that connects to the data source and fetches the same results for you.

Data-Driven Application: The "Write Code Yourself" Approach

In the last example, I posed a question: Where is the SQL Query?

I gave the answer along with the question—it is embedded deep inside autogenerated code. But why should a simple query such as that have to be embedded in so much code? It really doesn't have to be. As a matter of fact, when you do see the autogenerated queries in the strongly typed `DataSet` in Chapter 9, you'll see that the queries take an extra-safe approach by comparing all columns and specifying highly inefficient, but accurate, `UPDATE` and `DELETE` queries. This is because the autogenerated code must work in all situations and can't make any assumptions as it is written in a one-size-fits-all approach.

Usually, in any application architecture, you'll have to make a choice between performance, flexibility, and initial effort. Application architecture is a black art. Unfortunately or fortunately, you can't fully automate the process: unfortunately because it means more work for you and me, and fortunately because you and I will have jobs for a very long time. There are many instances where you'll have to consciously decide and pick between various approaches. That's what this book intends to help you do—give you enough knowledge so you can make those decisions intelligently as an application architect. As a matter of fact, generally in a full-blown enterprise application, you'll see yourself doing more hands-on work writing code yourself, rather than dragging and dropping.

Thus, it's important that the underlying data-access architecture gives you the ability, or fine-level control, to selectively choose what you need. Luckily, ADO.NET does give you this ability.

Let's look at a quick example that achieves the same results as the previous console application example, but this time around with no drag-and-drop help. You can download the code for this exercise from the associated code download, or you can easily create it using the following steps:

1. Create a new console application. Call it `ConsoleApp2`.

2. Add the necessary `using` or `Imports` statements at the top of `Program.cs` or `Module1.vb`:

   C#
   ```
   using System.Data.SqlClient ;
   ```

   VB.NET
   ```
   Imports System.Data.SqlClient
   ```

3. Let's cheat here a bit. You need a connection string to connect to the data source. Connection strings have been mentioned briefly earlier in this chapter, but they will be covered in more detail in Chapter 4. For now just copy and paste the connection string from any of the previous examples. Create the connection string as a private string, so it is accessible to the rest of the code within the class, but not outside of it, like so:

C#

```
private static string connectionString =
   "Data Source=(local);Initial Catalog=Test;Integrated Security=True";
```

VB.NET

```
Private connectionString As String = _
   "Data Source=(local);Initial Catalog=Test;Integrated Security=True"
```

4. With the connection string added, write in the following code in the `Main` function or subroutine:

C#

```
SqlConnection testConnection = new SqlConnection(connectionString);
SqlCommand testCommand = testConnection.CreateCommand() ;
testCommand.CommandText = "Select DemoValue from Demo where DemoID = 1" ;
testConnection.Open() ;
string result = (string)testCommand.ExecuteScalar() ;
testConnection.Close();
Console.WriteLine(result) ;
```

VB.NET

```
Dim testConnection As SqlConnection = New SqlConnection(connectionString)
Dim testCommand As SqlCommand = testConnection.CreateCommand()
testCommand.CommandText = "Select DemoValue from Demo where DemoID = 1"
testConnection.Open()
Dim result As String = CType(testCommand.ExecuteScalar(), String)
testConnection.Close()
Console.WriteLine(result)
```

5. Compile and run the application. You should see output as shown in Figure 3-21, which is very much like the output shown in Figure 3-20.

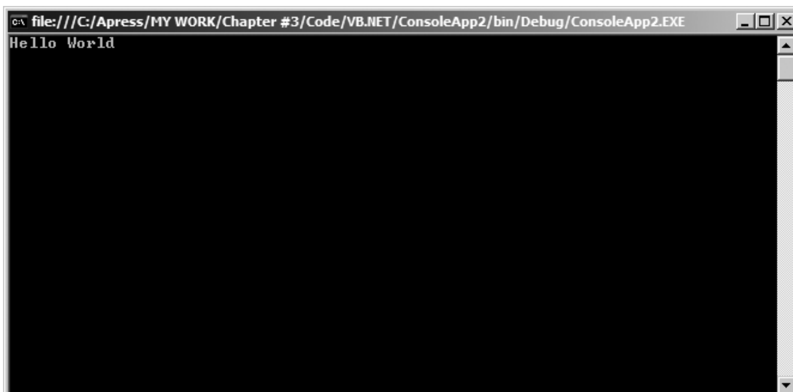

Figure 3-21. *The application that you wrote running in a hands-on way*

Here you were able to achieve the same results as the drag-and-drop approach in very few lines of code. By comparison, if you do actually browse the autogenerated code (see Chapter 9), you'll see that the autogenerated code can be hundreds or even thousands of lines of code.

Let's examine a bit more closely the code you just wrote. Some of it may not make sense yet (the necessary objects and methods involved are explained in Chapters 4 and 5), but let's look at the main steps involved:

1. First, *create an object that will hold the connection*. This is logical because to query the underlying data source you need a connection (this object is covered in detail in Chapter 4):

   C#
   ```
   SqlConnection testConnection = new SqlConnection(connectionString);
   ```

   VB.NET
   ```
   Dim testConnection As SqlConnection = New SqlConnection(connectionString)
   ```

2. Next, *create a command that will hold the SQL Query*. This could have even been a stored procedure, or it can even take parameters (the command object is covered in detail in Chapter 5):

   C#
   ```
   SqlCommand testCommand = testConnection.CreateCommand() ;
   testCommand.CommandText = "Select DemoValue from Demo where DemoID = 1" ;
   ```

   VB.NET
   ```
   Dim testCommand As SqlCommand = testConnection.CreateCommand()
   testCommand.CommandText = "Select DemoValue from Demo where DemoID = 1"
   ```

3. Next, in order to run the command, you need to *open the connection*:

   C#
   ```
   testConnection.Open() ;
   ```

   VB.NET
   ```
   testConnection.Open()
   ```

4. Now that the command is prepared and the underlying connection is open, you can *run the command* and fetch the results in a string variable:

   C#
   ```
   string result = (string)testCommand.ExecuteScalar() ;
   ```

   VB.NET
   ```
   Dim result As String = CType(testCommand.ExecuteScalar(), String)
   ```

5. Next, *close the connection*. This, as you will see in the next chapter, is extremely important to do. *Never fail to close an open connection*:

   C#
   ```
   testConnection.Close();
   ```

VB.NET

```
testConnection.Close()
```

6. With the results fetched in a string variable, you can now simply *show the results* using the `Console.Writeline` method:

C#

```
Console.WriteLine(result) ;
```

VB.NET

```
Console.WriteLine(result)
```

In brief, here are the steps you took to write a fully hand-written Hello World application:

1. Create a connection.
2. Create a command that holds the SQL Query.
3. Open the connection.
4. Run the command.
5. Close the connection.
6. Show the results.

When you see the steps without the objects involved, it seems like a fairly logical way to query your data source. Maybe the only curious thing to note is that the connection has been opened just before step 4—running the command—and closed immediately afterward. This golden rule of ADO.NET follows: "Open connections as late as possible and close them as early as you can."

This wraps up the last scenario presented in this chapter. In subsequent chapters, you'll come across the details of various objects involved that will help you create data-driven applications effectively.

Summary

Chapter 1 began by giving you a very high 30,000-ft. overview of what ADO.NET is, and where it fits into your general application architecture.

Chapter 2 got a little bit closer to the ground and took the discussion away from logical block diagrams and more toward class diagrams and the physical class structure layout of the various major objects and namespaces involved in ADO.NET. At this point, you should quickly glance back at Chapter 2 and compare the `SqlConnection` and `SqlCommand` objects involved in the last presented example in that chapter.

Chapter 3 was the first chapter in this book that presented you with true hands-on code, and you created four working data-driven applications.

So now that you have actually started walking on the ground of this new planet and have actually built some data-driven applications, it's time to start running by digging deeper into the framework. As an application architect and programmer, it's not only important to understand how to do something, but it's also important to understand how to do it right, and be able to reason between the various ways to achieve the same goal.

I can't tell you the one possible panacea simply because there isn't one that fits every single situation out there. This is because application architecture, especially ADO.NET, can't be zeroed down to black or white. There are too many scenarios to deal with and various shades of gray involved. However, certain things, such as not closing your database connections or abusing a `DataSet` as a database, are clearly bad.

Starting with Chapter 4, you'll see the various important objects involved in ADO.NET and the usage scenarios and best practices involved with them. As the book carries forward, the discussion will continue to visit various other important objects, their usage scenarios, and best practices involved.

Now that you have landed on this planet and have walked around a bit, tighten your seat belts, pack your bags, and hold onto your seat tightly, your magic carpet ride is about to begin with Chapter 4, "Connecting to a Data Source."