SYBASE iAnywhere

WHITE PAPER

Building ASP.NET Applications

with Delphi and Advantage Database Server

by Cary Jensen

TABLE OF CONTENTS

ABSTRACT

The Advantage Data Provider for .NET provides the most feature-rich mechanism for Delphi developers using the Advantage Database Server (ADS) to build ASP.NET Web applications and Web services. This paper begins with an examination of the data access options facing Delphi developers for application development using ADS, followed by a more detailed look at the benefits of the Advantage Data Provider in ASP.NET development. The remainder of this paper demonstrates some basic steps to ASP.NET development using Delphi and Advantage.

Due in part to Delphi's maturity as a database development environment, Delphi supports a wide range of data access options. Among these are ODBC drivers, OLE DB Providers, dbExpress drivers, Borland Data Providers (BDP) for .NET, as well as native TDataSet solutions, such as the Borland Database Engine (BDE).

Advantage also supports a wide range of data access options. Some of these are specific to a particular development environment, such as the Advantage TDataSet descendant for Delphi and the Advantage JDBC (Java) driver, while others are more generic, such as the Advantage Data Provider for .NET and the Advantage OLE DB Provider.

Together, this combination of options makes the choice of a data access mechanism for the Delphi developer using Advantage difficult. In most cases, however, it doesn't have to be. For most applications, there is a winning solution.

If you are a native Win32 Delphi developer, you will find the most compelling set of features in the Advantage TDataSet Descendant (Delphi 7 and older) or the Advantage Data Components (Borland Developer Studio and RAD Studio). Similarly, if you are building managed applications for the visual component library (VCL) for .NET, the same argument can be made for the Advantage TDataSet Descendant for .NET, which is also found in the Advantage Data Components. These downloads can be located on the Advantage Developer Zone site at http://devzone.AdvantageDatabase.com.

While the arguments for these data access solutions are strong, this paper is designed to specifically address Delphi ASP.NET development data access solutions. If you are interested in more details about Advantage's support for native Win32 Delphi or VCL for .NET, you can visit either the Advantage online help files or the Advantage Knowledgebase. These are both available on the preceding Web site.

Let's now turn our full attention to ASP.NET development with Delphi for .NET and the Advantage Database Server. Like Delphi's other development platforms, Delphi for .NET offers a wide range of data access options for ASP.NET developers. These include support for the generic foundation class library (FCL) data access classes (ODBC and OleDb), dbExpress drivers, BDP for .NET providers, as well as support for all third party solutions that support the ADO.NET data provider interface. (ADO.NET is the portion of the .NET FCL that applies to data access, storage, and manipulation.)

If understanding and choosing between these options weren't challenging enough, Delphi's own online help tends to favor dbExpress, for which there are only a limited number of drivers (none of which support Advantage), and sometimes even BDP for .NET, which is now deprecated. Furthermore, the tutorials in Delphi's help files often demonstrate the dragging and dropping of data access components from Delphi's Data Explorer as the starting point for ASP.NET development. Dragging and dropping from the Data Explorer is only supported by dbExpress drivers and BDP for .NET drivers, however.

Fortunately, the choice of data provider for ASP.NET development is often uncomplicated. In most cases, you should use the data provider supplied by the publisher of the database. For Advantage, this is Advantage Data Provider for .NET, which from this point forward, I'll simply refer to as the Advantage Data Provider.

In fact, the argument for the Advantage Data Provider is even stronger than that for most publisher-supplied data providers, when compared to generic solutions such as the FCL's OLE DB Provider. Specifically, because of the special capabilities of the Advantage Database Server itself, the Advantage Data Provider supports features unheard of in most other data providers. Examples of these features can be found in the AdsExtendedReader.

But there is a problem. You cannot begin a Web page by dragging and dropping the Advantage Data Provider from Delphi's Data Explorer. Since Delphi's help tends to emphasize this approach, Advantage developers sometimes don't know where to begin.

Before I continue with a demonstration of using the Advantage Data Provider in Delphi ASP.NET applications, let me first address the Data Explorer drag and drop approach. To put it simply, this is not a technique that I recommend. While this approach makes it very easy to create a simple ASP.NET Web page, it offers very little in the way of code reuse.

In short, when you place your data access components directly on your Web pages (which is what you are doing when you drag and drop), you are coupling your user interface and your data access mechanism, something that many developers prefer to avoid. I'll admit, there are some benefits to this coupling, including an ease with which visual Web controls can be bound to the underlying data.

However, the disadvantages are significant. Primary among these is the lure of encapsulating data access logic, such as query definitions and stored procedure calls, within individual Web pages. If your site consists of just a few Web pages, this is not a problem. However, most Web sites consist of dozens of pages, and sometimes hundreds.

For most larger Web sites, hiding data access specifics on individual Web pages introduces debugging and maintenance issues. For example, if you subsequently change the underlying structure of your database or the parameters of your stored procedures, you need to search through countless Web pages looking for queries and stored procedure calls that are affected by those changes. This is especially frustrating if the same basic query appears on many different Web pages. Each duplicate must be located, and each needs to be modified in the same manner.

Ideally, a class, or set of classes, independent of the Web pages on which their data is displayed, should perform data access. I like to call this class, or set of classes, a reusable data layer, and I have written about this approach in the past. For a detailed white paper on creating a reusable data layer for the Advantage Data Provider, use this URL: http://www.sybase.com/detail?id=1054244.

(Author's note: While the aforementioned paper demonstrated an abstract data layer, it did so in a very generic way. So generic, in fact, that the SQL statements in the example were embedded in the individual Web pages. In practice, an abstract data layer encapsulates the queries, stored procedure calls, and views that embody the data access logic. Individual Web pages often pass parameters to the exposed methods of the abstract data layer, and these affect the results that are returned, as opposed to passing entire query strings, which in part, defeats the de-coupling that an abstract data layer can provide.)

Instead of focusing on the reusable data layer approach, the remainder of this paper sheds light on some of the components of the Advantage Data Provider that you use to access your Advantage data. Due to space limitations, this discussion will focus on the primary subset of components and techniques. However, it should be enough to get you started.

For more detailed information on using classes of the Advantage Data Provider, please refer to the Advantage help files. You may also want to reference Delphi's help and other .NET resources for more detailed explanations of the use of these and other classes available in ADO.NET.

GETTING STARTED

Before we discuss the principle classes of the Advantage Data Provider that you use in your applications, we should take a few moments to consider how you prepare your Delphi ASP.NET application to use Advantage. While these steps are not complex, they are essential.

First, you must begin with a Delphi for .NET ASP.NET application. The following steps assume you are using CodeGear's RAD Studio 2007. The steps are similar if you are using Borland Developer Studio (BDS) 2006 or 2005. Furthermore, while it is not directly referenced in this paper, and the exact steps to generate a project are different, the data layer abstraction technique described applies to ASP.NET applications written in Delphi Prism as well.

Begin by starting Delphi in a mode that includes the .NET personality. If you have installed the full version of RAD Studio, you can simply load RAD Studio. It includes all personalities. Alternatively, you can select the Window's Start Menu, select the CodeGear folder, and select either RAD Studio or Delphi for .NET. Delphi for .NET includes only the .NET personality.

Note that it is possible to install RAD Studio with some, but not all three of RAD Studio's personalities. If you did not install the Delphi for .NET personality, you need to re-install Delphi, this time ensuring that the .NET personality is selected for installation.

With Delphi running, select File | New | ASP.NET Web Application from Delphi's main menu.

When the New ASP.NET Web Application dialog box appears, as shown in the following figure, enter the name of your application. Note that not only will this name be used for your project source file, but it will also be used to create a virtual directory in the folder c:\inetpub\wwwwroot. For the sake of this example, enter the name AdsASPNETApplication. Click OK to continue.

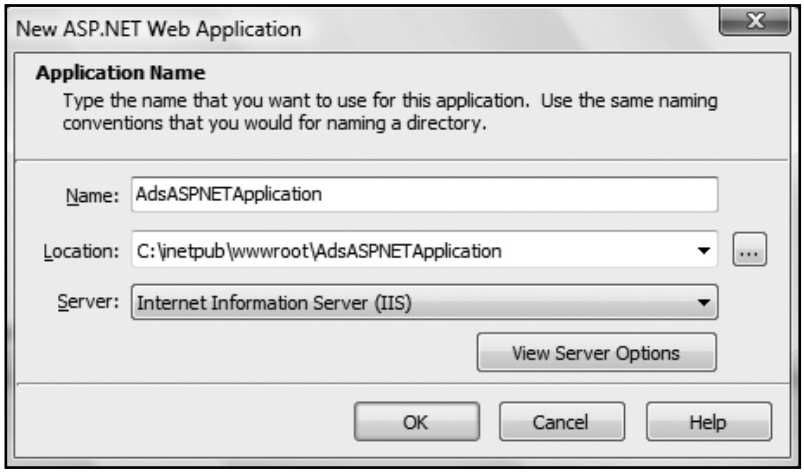

Figure 1

You now have a new ASP.NET application, and can begin entering text, HTML, as well as adding Web controls. However, before you can begin accessing Advantage, you have to add the Advantage Data Provider assembly to the project's references folder.

To do this, right-click the References folder in the Project Manager, and select Add Reference.

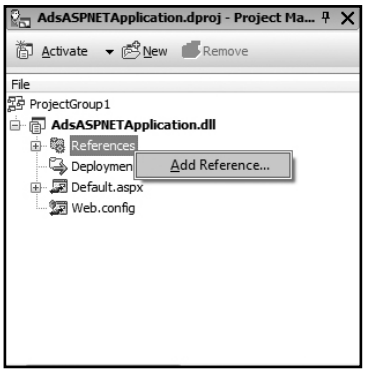

Figure 2

Delphi displays the Add Reference dialog box, which will take a few moments to be populated. You do not have to wait for the .NET Assemblies tab to be populated. Simply click the Browse button on the right side of this dialog box and navigate to the directory where the Advantage Data Provider is installed. By default, this location is c:\Program Files\Advantage\ado.net\2.0 (1.0 if you are using BDS 2006 or BDS 2005, since they use ASP.NET 1.1).

From this folder, select Advantage.Data.Provider.dll and click Open. Once selected, the assembly appears in the lower portion of the Add References dialog box, as shown here.

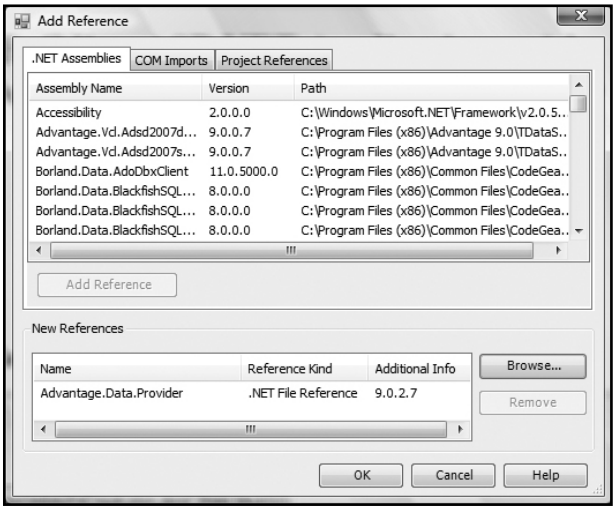

Figure 3

When you are done, click OK. Your References folder should now look something like the following.

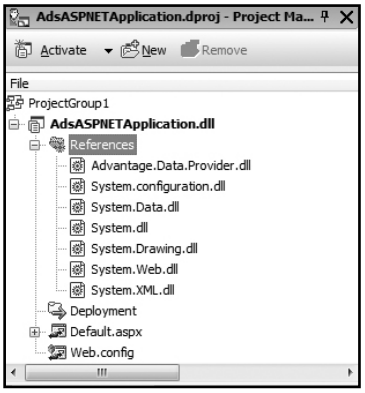

Figure 4

This next step is optional. My personal preference is to use the Advantage Data Provider assembly as a local copy, which means that a copy of the assembly will be copied to the bin subdirectory of your application's directory. This permits you to deploy your ASP.NET application to your Web server by copying just the application directory structure without having to install any other files, other than the .NET framework itself, of course.

There are simply too many issues associated with this topic to cover here, so I will let you make your own mind up about this step. However, if you want to follow my suggestion, you need to right-click the Advantage.Data.Provider.dll assembly node in the References folder of Delphi's Project Manager and select Copy Local from the displayed menu. Delphi will display a warning, but that's ok. Select Yes to accept the local copy designation.

There is only one more step before you are ready to go. You must add the Advantage.Data.Provider namespace to the uses clause of any unit from which you want to reference any of the classes of the Advantage Data Provider. The following is an example of a uses clause where this namespace appears.

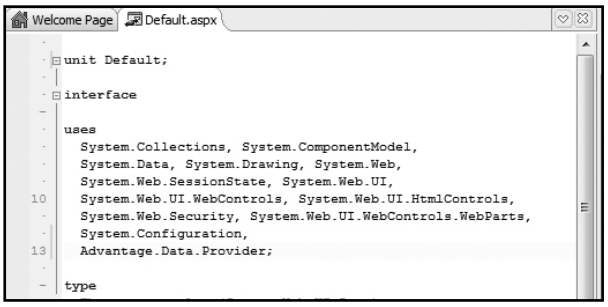

Figure 5

You are now ready to reference and use the classes of the Advantage Data Provider.

The Primary Classes of the Advantage Data Provider

As an Advantage user, the primary classes that you will use in your ASP.NET applications include AdsConnection, AdsCommand, AdsDataReader, AdsExtendedReader, and AdsDataAdapter. There is also a good chance that you'll use AdsTransactions and AdsParameters. I'll begin by briefly describing each of these classes.

You use an AdsConnection to make your connection to the Advantage Database Server. The ConnectionString property of AdsConnection defines where your data is located, user and password information, and other specifics of the connection, including the size and duration of its connection pool.

Once a connection is established, you use one or more AdsCommands to configure your database as well as access its data. AdsCommands can execute statements that return no result sets (such as a CREATE TABLE or DROP VIEW statement), return a scalar value (a single value of TObject type), or a result set. When a result set is returned, it can be navigated using a DataReader, or can be used to populate an in-memory DataTable.

An AdsDataReader is a read-only, forward-only cursor that you can use to read the records returned by a SELECT statement or a stored procedure that returns one or more records. In addition, the Advantage .NET Data Provider supports a super-charged data reader in the form of the AdsExtendedReader.

The AdsExtendedReader is unique in the .NET world, in that it provides bi-directional navigation, as well as read/write access to data. In addition, it supports server-side cursors for high-speed navigation as well as optimized filtering and scopes (also known as ranges).

The AdsDataAdapter is a component that uses an AdsCommand internally to load data from a result set into an in-memory DataTable. DataTables are useful for loading data into grid-like controls, and also provide a convenient mechanism for persisting result sets between Web page renderings.

When your code needs to provide all-or-none modifications to two or more records, or data in two or more database tables, you use an AdsTransaction. You use the BeginTransaction method of your connected AdsConnection to create an AdsTransaction. Following your database changes, you call either the AdsTransaction's Commit or Rollback methods to perform the associated operation.

When the CommandText property of your AdsCommand specifies a parameterized query or stored procedure call, you use instances of AdsParameters to define the data types, names, and values of those parameters. These AdsParameters can be created by calling the AdsCommand's CreateParameter method, or by calling the AdsParameter's constructor. Once the created parameters are configured, they are added to the AdsCommand's Parameters collection property.

Some Basic Operations with the Advantage Data Provider

This section demonstrates some basic operations using the classes of the Advantage Data Provider. The sample database and data dictionary used in these examples is described in the book *Advantage Database Server: A Developer's Guide* (Sybase, Inc., 2007), written by myself and Loy Anderson.

The complete text of this book is available in the Advantage 9.0 Help files. The following examples assume that you have followed the steps through Chapter 4 of the book to create a data dictionary from the tables supplied in the book's sample applications and code. See Appendix A for instructions on downloading the sample applications and source code.

You establish a connection to a data dictionary, or to a folder containing free tables, using an AdsConnection. The following code segment demonstrates a simple connection to the data dictionary of the sample database.

```
…
var
  Connection: AdsConnection;
begin
  Connection := AdsConnection.Create;
  Connection.ConnectionString := 'Data Source='+
    'C:\AdsBook\DemoDictionary.add;user id=adssys;password=password;' +
    'ServerType=ADS_REMOTE_SERVER | ADS_LOCAL_SERVER;';
  Connection.Open;
  try
    //code to use the connection here
  finally
    Connection.Close;
  end;
end;
```

This simple example demonstrates calling the constructor of an AdsConnection, assigning its connection string, followed by a call to open the connection. It also demonstrates the call to close the connection. In all of the following examples, it is assumed that this connection is active, and referred to by the variable named Connection.

In this example, the connection string is represented literally in the code. Note, however, that it is also possible to store the connection string in the web.config file (the application configuration file) or in machine.config (a configuration file available to all applications on the Web server in ASP.NET 2.0 and later).

In ASP.NET applications, it is very important to ensure that each call to open a connection is matched by a corresponding call to close the connection. The Advantage Data Provider gets its connections from a connection pool, by default. If you fail to call close, the connection will not be returned to the connection pool until the ASP.NET garbage collector destroys the connection (and this might take a while). By closing the connection explicitly, you ensure that the connection returns to the pool following its use, making it immediately available for use by another Web page.

You use the connection to create AdsTransactions and AdsCommands. Let's consider AdsCommands first.

The following code segment creates a table named #MyTable (the # character designates this table as a temporary table, one that exists only for the duration of the current connection).

```
var
  Command: AdsCommand;
begin
  Command := Connection.CreateCommand;
```

```
  Command.CommandText :=
            'CREATE TABLE #MyTable (' +
                                      'NumField Integer, ' +
                                      'StringField cichar ( 100 ))';
  Command.ExecuteNonQuery;
```

The CREATE TABLE command is a SQL data definition language (DDL) command, and it does not return a value or a result set. If your query returns a scalar value, you can use ExecuteScalar, which returns a TObject result. It is necessary to convert or cast this result to the appropriate data type, as demonstrated in the following code segment.

```
var
  Command: AdsCommand;
  IntValue: Integer;
begin
  Command := Connection.CreateCommand;
  Command.CommandText := 'SELECT Count(*) FROM #MyTable';
  IntValue := Convert.ToInt32(Command.ExecuteScalar);
```

Whenever you need to perform two or more operations, and have them succeed in an all-or-none fashion, you use an AdsTransaction. The following code demonstrates the insertion of four records into the temporary table named #MyTable from within a transaction.

```
var
  Command: AdsCommand;
  Transaction: AdsTransaction;
begin
  Transaction := Connection.BeginTransaction;
  try
    Command := Connection.CreateCommand;
    Command.CommandText := 'INSERT INTO #MyTable VALUES (1, ''one'');';
    Command.ExecuteNonQuery;
    Command.CommandText := 'INSERT INTO #MyTable VALUES (2, ''two'');';
    Command.ExecuteNonQuery;
    //The following command contains a SQL Script
    Command.CommandText :=
        'INSERT INTO #MyTable VALUES (3, ''three'');' +
        'INSERT INTO #MyTable VALUES (4, ''four'');';
    Command.ExecuteNonQuery;
    Transaction.Commit;
  except
    Transaction.Rollback;
  end;
```

When your query returns a result set, you can either use a data reader to obtain a cursor to the result set, or you can load the result set into an in-memory DataTable. The following example demonstrates a function that returns a result set using an AdsDataReader.

```
function ReusableDataLayer.GetAllEmployeeNames: AdsDataReader;
var
  Command: AdsCommand;
begin
```

```
  Command := Connection.CreateCommand;
  Command.CommandText := 'SELECT [First Name] + '' '' +
                         '[Last Name] as Name, ' +
                         '[Employee Number] FROM [EMPLOYEE]';
  Result := Command.ExecuteReader;
end;
```

This next code demonstrates how the preceding code, which is associated with a reusable data layer, can be used to bind data to a list box.

```
procedure TUseDataReader.Button1_Click(sender: System.Object;
                                       e: System.EventArgs);
var
  ReusableDataLayer1: ReusableDataLayer;
  DataReader: AdsDataReader;
begin
  ReusableDataLayer1 := ReusableDataLayer.Create(Context);
  try
    DataReader := ReusableDataLayer1.GetAllEmployeeNames;
    try
      ListBox1.DataSource := DataReader;
      ListBox1.DataTextField := 'Name';
      ListBox1.DataValueField := 'Employee Number';
      DataBind;
    finally
      DataReader.Close;
    end;
  finally
    ReusableDataLayer1.Dispose;
  end;
end;
```

Please note that in the preceding code, the Dispose method of the reusable data layer calls the AdsConnection's Close method. In addition, it is important to ensure that you close a data reader when you are through with it.

The following figure shows how your ListBox might look after having been populated by the preceding code.

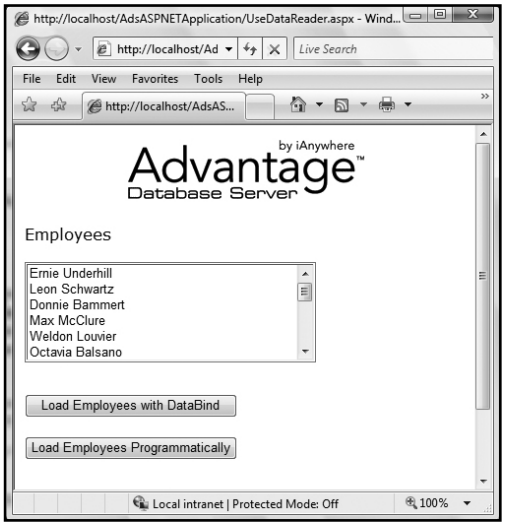

Figure 6

While the preceding code demonstrates using data binding to load the data from the data reader into the list box, the same effect could also be achieved by loading the list box programmatically. This requires you to iterate through the data reader, loading each row into the list box manually. The following code performs exactly the same task as the preceding code, without the call to DataBind.

```
procedure TUseDataReader.Button2_Click(sender: System.Object;
                                       e: System.EventArgs);
var
  ReusableDataLayer1: ReusableDataLayer;
  DataReader: AdsDataReader;
begin
  ReusableDataLayer1 := ReusableDataLayer.Create(Context);
  try
    DataReader := ReusableDataLayer1.GetAllEmployeeNames;
    try
      while DataReader.Read do
        ListBox1.Items.Add(ListItem.Create(DataReader.GetString(0),
                           DataReader.GetInt32(1).ToString));
    fibnally
      DataReader.Close
    end;
  finally
    ReusableDataLayer1.Dispose;
  end;
end;
```

DataTables are classes that you use to hold part or all of a result set. These values are held in memory, and can be used for data manipulation, data binding, and even storing sets of data for later use. One of the easiest ways to populate a DataTable is through the use of an AdsDataAdapter.

An AdsDataAdapter contains an AdsCommand that returns a result set, either through a SQL SELECT statement or a call to a stored procedure that returns a result set. Once you configure the AdsDataAdapter, you call its Fill method to transfer this data into the DataTable. This is demonstrated in the following method, which returns a DataTable. This code also demonstrates the use of an AdsParameter.

```
function ReusableDataLayer.GetCustomerSales(CustNo: Integer):DataTable;
var
  Command: AdsCommand;
  Parameter: AdsParameter;
  DataAdapter: AdsDataAdapter;
begin
  Command := Connection.CreateCommand;
  Command.CommandText := 'SELECT * FROM INVOICE ' +
                         '  WHERE [Customer ID] = :custno';
  Parameter := Command.CreateParameter;
  Parameter.DbType := System.Data.DbType.Int32;
  Parameter.ParameterName := 'custno';
  Parameter.Value := TObject(CustNo);
  Command.Parameters.Add(Parameter);
  DataAdapter := AdsDataAdapter.Create(Command);
```

```
  Result := DataTable.Create;
  DataAdapter.Fill(Result);
end;
```

The following code uses the preceding method. The value that is passed to the GetCustomerSales method is obtained from a TextBox on a Web page.

```
procedure TDataTable.Button1_Click(sender: System.Object;
                                   e: System.EventArgs);
var
  ReusableDataLayer1: ReusableDataLayer;
  DataReader: AdsDataReader;
begin
  ReusableDataLayer1 := ReusableDataLayer.Create(Context);
  try
    DataGrid1.DataSource :=
      ReusableDataLayer1.GetCustomerSales(
                          Convert.ToInt32(TextBox1.Text));
    DataBind;
  finally
    ReusableDataLayer1.Dispose;
  end;
end;
```

The following is an example of how the Web page looks after the preceding code executes.

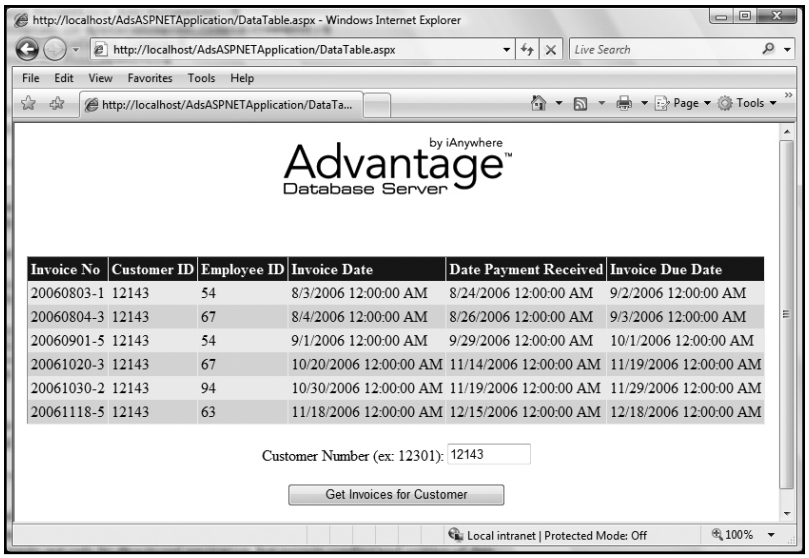

Figure 7

The final example demonstrates the use of an AdsExtendedReader. As mentioned earlier in this paper, the AdsExtendedReader is unique in the world of .NET data providers, in that it permits not only bi-directional navigation, but permits reading and writing of data as well.

```
procedure ReusableDataLayer.InsertAndUpdate;
  var
    Command: AdsCommand;
    ExtendedReader: AdsExtendedReader;
    Transaction: AdsTransaction;
  begin
    Transaction := Connection.BeginTransaction;
    try
      Command := Connection.CreateCommand;
      Command.CommandText :=
          'SELECT * FROM ITEMS ' +
          '  WHERE [Invoice No] = ''20061130-2'' ' +
          '    AND [Product Code] = ''H54039''';
      ExtendedReader := Command.ExecuteExtendedReader;
      try
        if ExtendedReader.Read then
          if ExtendedReader.GetInt32(
               ExtendedReader.GetOrdinal('Quantity')) = 1 then
          begin
            ExtendedReader.SetInt32(
              ExtendedReader.GetOrdinal('Quantity'), 2);
            ExtendedReader.WriteRecord;
          end;
      finally
        ExtendedReader.Close;
      end;
      Command.CommandText :=
          'SELECT * FROM ITEMS ' +
          '  WHERE [Invoice No] = ''20081105-1'' ' +
          '    AND [Product Code] = ''H54050''';
      ExtendedReader := Command.ExecuteExtendedReader;
      try
        if not ExtendedReader.Read then
        begin
          ExtendedReader.AppendRecord;
          ExtendedReader.SetString(0, '20081105-1');
          ExtendedReader.SetString(1, 'H54050');
          ExtendedReader.SetInt32(2, 1);
          ExtendedReader.SetInt32(3, 0);
          ExtendedReader.SetDecimal(4, 199.99);
          ExtendedReader.WriteRecord;
        end;
      finally
        ExtendedReader.Close;
      end;
      Transaction.Commit;
    except
      Transaction.Rollback;
    end;
  end;
```

Without the use of an AdsExtendedReader, you would have to construct the UPDATE query to update the first record modified by this example, as well as use a SELECT query (to detect if the second record already exists) followed by an INSERT query.

I simply cannot overemphasize the value of the AdsExtendedReader in ASP.NET applications. While you can always rely on SQL statements with Advantage to perform your updates, inserts, deletes, and selections, the AdsExtendedReader provides you with an alternative that, in some cases, saves you from writing dozens, if not hundreds of lines of code.

SUMMARY

The Advantage Data Provider for .NET is the preferred data access mechanism for developing ASP.NET applications with Delphi and the Advantage Database Server. The Advantage Data Provider is easy to use and deploy, and provides you with advanced features unavailable in any other .NET data provider.

ABOUT THE AUTHOR

Cary Jensen is President of Jensen Data Systems, Inc., a Texas-based company that specializes in Internet and database development, training, and consulting. He is an authority on database development and is an award-winning, best-selling author of 20 books, including the *Advantage Database Server: A Developer's Guide* (Sybase, Inc., 2007), *Advantage Database Server: The Official Guide* (McGraw-Hill), *Building Kylix Applications* (Osborne/McGraw-Hill), *JBuilder Essentials* (Osborne/McGraw-Hill), *Delphi In Depth* (Osborne/McGraw-Hill), *Oracle JDeveloper* (Oracle Press), and *Programming Paradox 5 for Windows* (Sybex).

Cary is the author and speaker for the Advantage Developer Days seminars and the Delphi Developer Days seminars. He speaks at conferences, workshops, and training seminars throughout North America and Europe. Cary has a Ph.D. from Rice University in Human Factors Psychology, specializing in human-computer interaction.

Contact Information

Cary Jensen
Jensen Data Systems, Inc.
Phone: 281-359-3311
Email: CJensen@JensenDataSystems.com
Web Site: http://www.JensenDataSystems.com

Sybase, Inc.
Worldwide Headquarters
One Sybase Drive
Dublin, CA 94568-7902
U.S.A
1 800 8 sybase

www.sybase.com/ianywhere

SYBASE® iAnywhere®