Chapter 18
ADS and ADO.NET

Note: This chapter accompanies the book Advantage Database Server: A Developer's Guide, 2nd Edition, by Cary Jensen and Loy Anderson (2010, ISBN: 1453769978). For information on this book and on purchasing this book in various formats (print, e-book, etc), visit: http://www.JensenDataSystems.com/ADSBook10

Things change, and in the computer software industry things change fast. But one thing that hasn't changed is your need to access data. It's the way you access data that has changed.

One of the more recent data access mechanisms, at least for now, is ADO.NET, the data access layer of the .NET FCL (framework class library). This technology is significant, and if you are not already using it, you may be some day in the future. And if you do, you'll be glad to find that Advantage is there with you.

This chapter provides you with an introduction to data access using the .NET FCL, specifically ADO.NET. It begins with an overview of ADO.NET and the Advantage .NET Data Provider. It continues with a look at accessing your Advantage data using Advantage and ADO.NET.

In keeping with the language-agnostic nature of Advantage, data access is demonstrated in this chapter using C#, one of the newer object-oriented languages. Keep in mind, however, that .NET classes are .NET classes, regardless of which .NET-enabled language you use. Consequently, all the data access techniques demonstrated in this chapter can be used from any .NET language, including VB for .NET, Delphi Prism, and Visual J# for .NET, to name a few.

Note: Delphi 2005 through 2007 also supported .NET development, though this capability has since been deprecated, with Delphi Prism providing the replacement technology. The Advantage .NET Data Provider for Advantage 10 does not support these older versions of Delphi for .NET. If you are supporting a Delphi for .NET project in Advantage, you can use the Advantage .NET Provider for Advantage 9.1. Since Advantage clients tend to be forward compatible, you can use this provider with Advantage 10.

There is an additional data access framework in .NET that builds upon ADO.NET. This framework, which is often referred to as the Entity Framework, uses a layer of abstraction between your applications and the lower-level .NET components. While this chapter provides you with detail coverage of these lower-level classes, you will find an in-

depth look at the Entity Framework in Chapter 19, *Advantage and the .NET Entity Framework*.

Advantage and ADO.NET

ADO.NET is the common name for the classes and interfaces of the System.Data second-level namespace of the FCL. Conceptually, ADO.NET can be divided into two distinct parts: the data access layer and the data storage system.

The classes associated with the data storage system are stand-alone classes that you can employ in any ADO.NET application. These classes include DataColumn, DataRelation, DataRow, DataSet, DataTable, and DataView. Of these, the most central class is DataSet.

Unlike the storage mechanism, which is defined around classes, the data access layer is formally defined using interfaces (abstract application programming interface definitions), and in ADO.NET 2.0 and later, abstract base classes. Concrete classes—that is, classes that can be instantiated—implement these interfaces. It is these concrete implementations that you use to access your data. Such classes are referred to generically as .NET data providers.

There are five .NET data providers native to ADO.NET, which is also to say that they are installed along with the FCL. These are associated with the System.Data.SqlClient, System.Data.OleDb, System.Data.Odbc, System.Data.SqlServerCE, and System.Data.OracleClient third-level namespaces.

As is the case with ADO and OLE DB providers, database vendors are permitted and encouraged to create their own data access classes that implement these same interfaces (and in ADO.NET 2.0, extend the abstract base classes). Advantage calls their implementation the Advantage .NET Data Provider, and it can be found in the Advantage.Data.Provider namespace.

The primary responsibility of the Advantage .NET Data Provider classes is the same as that of the native implementations—supply data to the data storage system, and in particular, provide access to data through SQL queries.

The second, and obviously crucial, role of .NET data providers is to permit direct manipulation of data through SQL. This, too, is deftly handled by the Advantage .NET Data Provider.

As is the case with all data access mechanisms described in this part of this book, the following discussion of Advantage programming with ADO.NET touches on just a few of the available techniques, particularly those that apply to Advantage. Unlike some of the other data access mechanisms covered in this part of this book, the native .NET classes provide a significant amount of standard database functionality, such as filtering, sorting, and seeking. Consequently, these topics are not Advantage specific, and are not covered in this chapter. For a comprehensive discussion of ADO.NET programming, you may want to pick up a book on the subject.

Another important point about the examples provided in this chapter is that they are provided through a Windows form application. This might seem a bit odd, considering that the vast majority of applications written using the .NET framework are Web forms (or Web service applications), those used to create applications for the Web. (Though Silverlight-based applications are increasing in popularity.)

There is a good reason for the use of a Windows form application in these examples. Windows forms applications are more demanding, database-wise, than ASP.NET applications, such as Web forms applications. In most Web forms applications, a DataReader is used to read data, which is then bound to one or more controls on a Web form. In most cases, DataSets and DataTables are not used. These are, however, essential parts of ADO.NET.

There is also a second reason. Creating an ASP.NET Web forms application is more involved than creating a Windows forms application. You need a Web server and you need to configure it to run .NET.

In the end, we opted for focusing on how to access Advantage using .NET, rather than complicate the process with ASP.NET-related issues. Consequently, the Windows forms application described here is designed to be similar to the Delphi, Java, Visual Basic, and Visual FoxPro applications described in other chapters in this part.

The Right Provider for Visual Studio

There is little doubt that .NET will be an important framework for many years to come. As a result, it is not surprising that the Advantage team has committed itself to providing Advantage developers with an exceptional .NET experience.

To put this another way, the Advantage team has provided specific customizations that integrate Advantage beautifully with Visual Studio. This includes the availability of custom component editors in the Visual Studio IDE (integrated development environment), support for ADO.NET 2.0, as well as access to your database metadata from within the Visual Studio Server Explorer. Each of these points is discussed in the following sections.

Visual Studio Component Editors

Advantage provides exceptional design time support in Visual Studio. For example, if you choose to configure your data access at design time, Advantage gives you the automated tools you need to configure your components. When you drop an AdsDataAdapter onto a design surface, Advantage launches the Advantage Data Adapter Configuration Wizard. You can also manually invoke this wizard by right-clicking an AdsDataAdapter and selecting Configure Data Adapter.

This wizard, shown in Figure 18-1, provides you with support for both building your connection string and creating the SQL statements that you associate with the data adapter's DeleteCommand, InsertCommand, SelectCommand, and UpdateCommand

properties. In many cases, you only need to add the final code to open your AdsConnection and fill your datasets.

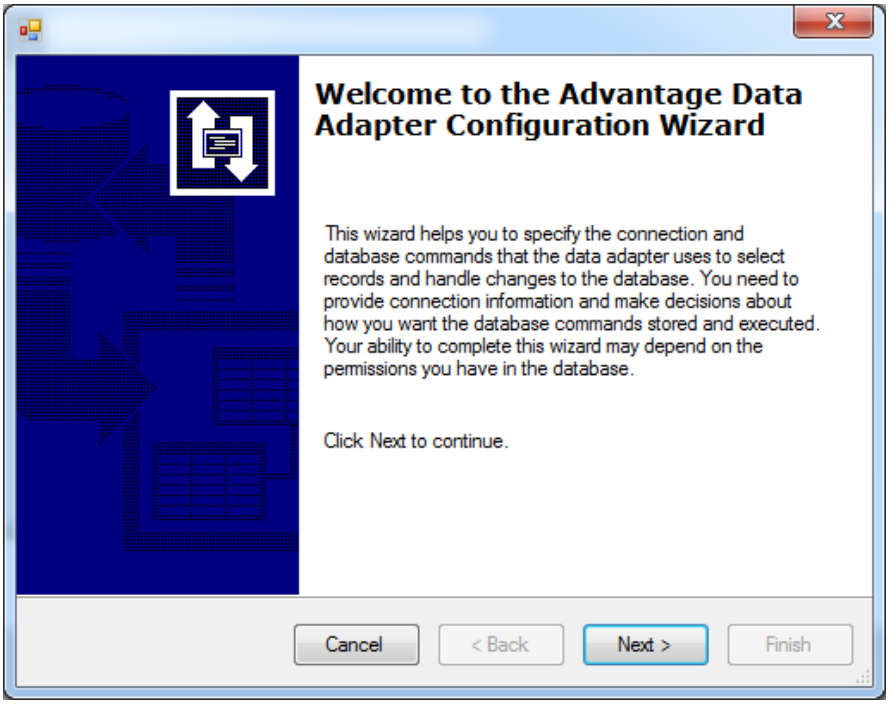

Figure 18-1 The Advantage Data Adapter Configuration Wizard

Likewise, if you right-click the AdsDataAdapter and select Generate Dataset, a Component Editor appears that helps you configure a DataSet to use the result set generated by the AdsDataAdapter.

Finally, the AdsDataAdapter class also provides you with a special, integrated viewer that permits you to inspect your data directly within Visual Studio. Simply right-click a configured AdsDataAdapter and select Preview Data to access this feature.

If instead of using design-time configuration, you can choose to create and configure your data access components at runtime (which you will want to do if your applications need to be scalable) and still use the code generated by the Advantage Data Adapter Configuration Wizard. Simply place an AdsDataAdapter into a form and use the wizard to configure the adapter and its AdsCommand. Then, copy the generated code and paste it into the appropriate location of your runtime code project.

Full ADO.NET 2.0 Support

Advantage is one of the .NET data providers to fully support the ADO.NET 2.0 framework, which represented a major departure from the previous architecture employed in ADO.NET 1.1 and earlier. To begin with, the primary Advantage .NET data provider classes descend from the ADO.NET base classes, including DbConnection, DbCommand,

DbDataAdapter, and so forth. Furthermore, Advantage provides an implementation of the DbFactory class, AdsFactory, which you can use to write more portable code.

Note: The core architecture of ADO.NET has not changed since ADO.NET 2.0. What has changed, in .NET 3.0, 3.5, and the latest, .NET 4.0, is the addition of language integrated query (LINQ) capabilities, which introduces an additional layer of abstraction between your data providers and your application. LINQ, and Entity Framework in particular, are covered in Chapter 19, Advantage and the .NET Entity Framework.

Additional ADO.NET features supported by Advantage include support for ambient transactions through the System.Transaction.TransactionScope class. Using a TransactionScope, all subsequent connections to Advantage through the Advantage .NET Database Provider enlist the services of the TransactionScope, which you can then use to commit or roll back updates to your data.

Another new class in the Advantage .NET 2.0 Data Provider is the AdsConnectionStringBuilder. This class provides your code with help building a connection string by exposing properties that map to the connection string parameters. After setting its properties, you can read the correctly formatted connection string from the AdsConnectionStringBuilder's ConnectionString property.

You can also assign a valid connection string to the ConnectionString property, and then read the individual connection parameters from the AdsConnectionStringBuilder's properties.

In a similar vein, Advantage provides a class that helps you create and use the various classes of ADO.NET without having to create these classes directly. This class, the AdsHelper, can be used to create AdsCommands, execute queries, and fill Datasets. This class is available in both C# and VB for .NET source code that is installed in the Advantage .NET Data Provider installation directory.

You can add this source file directly to your projects, or you can compile a class library containing this class (which you then add to your References folder).

Note: For information on using the AdsHelper class, see the document AdsHelper.pdf, which is located in the installation directory of the Advantage Data Provider.

While all of this is very nice, it’s the Advantage Extended reader that we really love. This class, AdsExtendedReader, provides all of the basic functionality described by the IDataReader interface. But it does so much more.

While all data readers permit you to read data and navigate forward in a result set, the AdsExtendedReader adds support for server-side cursors. By leveraging the unique character of Advantage's architecture, the AdsExtendedReader provides you with read and write capabilities, bi-directional navigation; and server-side, optimized seeks, scopes, and filters. In our ASP.NET Web forms and Web services applications, we have found these

capabilities to greatly simplify the process of working with data. We cannot imagine ADO.NET without the extended reader, but that is the world in which other .NET developers live.

Advanced Visual Studio Integration

With Visual Studio 2005 and later, Advantage's integration goes even further. You can install Advantage data connections into the Visual Studio Server Explorer. With Advantage data connection added to the Server Explorer, you can easily view your database's metadata from within Visual Studio, including the names of tables, views, and stored procedures, as well as field and parameters names and data types.

Figure 18-2 shows a data connection to the DemoDictionary data dictionary in the Server Explorer found in Visual Studio 2010.

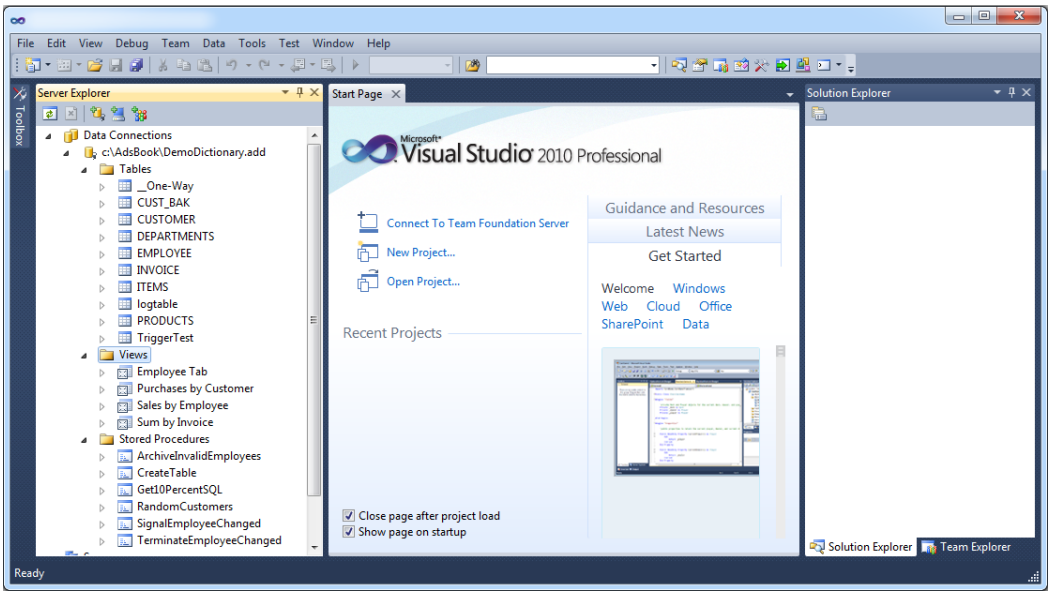

Figure 18-2: You can access your Advantage data from the Visual Studio 2005 Server Explorer

A Note About the Examples

The classes of the Advantage .NET Data Provider are instantiated in these examples at runtime rather than being placed and configured at design time. While Visual Studio permits you to place and configure data access components such as AdsConnection, AdsDataAdapter, and DataSet at design time, we do not recommend this technique.

Some developers disagree with our approach. These developers correctly point out that the code generated by IDEs when you place and configure your data access components at design time saves you a lot of time coding and simplifies the maintenance of your applications.

Our position, and one that is shared by many in the .NET database community, is that when you use design-time placement and configuration, you have little control over the generated code, and the results may scale poorly. But whether or not you agree with this approach, the examples in this chapter do a good job of demonstrating how to create, configure, and control ADO.NET-related classes programmatically.

Code Download: The examples provided in this chapter can be found in the C# project CS_ADONET_2010 and CS_ADONET_2008 available with this book's code download (see Appendix A). These projects were written in Visual Studio 2010 and Visual Studio 2008, respectively. If you are working with an older version of Visual Studio, project format incompatibilities will prevent you from compiling these projects, though you could still easily reuse much, if not all, of the code.

Note: Visual Studio 2008-2010 can compile .NET projects as either 32-bit or 64-bit assemblies. By default, your project's Platform target options are set "any CPU," which will create 32-bit applications if your operating system is 32-bit and 64-bit applications if your operating system is 64-bit. If you are using a 64-bit Windows operating system, and this application throws an exception when you attempt to load the Advantage Data Provider, you may need to adjust this project's platform to x86, or see the Advantage Knowledgebase entry that describes how to work around a bug in .NET to permit your application to load Advantage's 64-bit client.

The main form of the CS_ADONET C# project used in the examples in this chapter is shown in Figure 18-3.

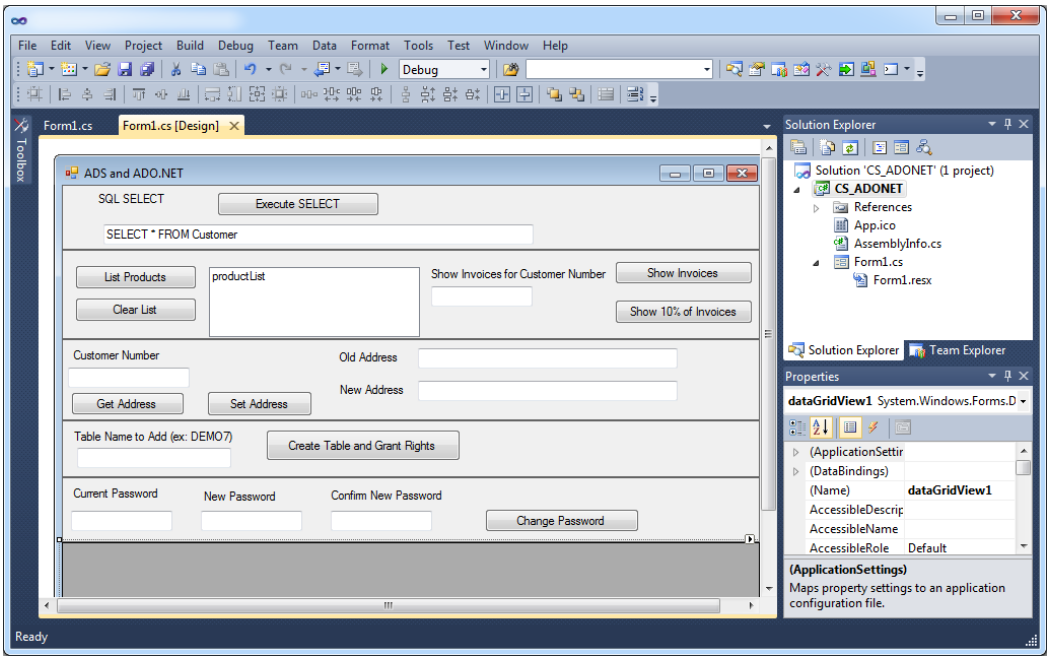

Figure 18-3: The CS_ADONET project in Visual Studio 2010

Note that before you can use Advantage with Visual Studio for .NET, you must add a reference to the Advantage.Data.Provider assembly. To do this in Visual Studio 2010, use the following steps:

1. From a Visual Studio .NET project, right-click the References folder in the Solution Explorer.
2. Select Add Reference.
3. Use the displayed dialog box to select the Advantage.Data.Provider assembly from the .NET tab.
4. Click OK when you are done.

Note: If you place your components into your projects at runtime from the Tool Palette in Visual Studio, Visual Studio will add the Advantage.Data.Provider assembly to the References folder of you project for you. However, you must first manually add your Advantage components to the Tool Palette. To do this in Visual Studio 2010, select Tools | Choose Toolbar Items. Once the resulting dialog box is displayed (which often takes a bit of time), place a checkmark next to the AdsCommand, AdsCommandBuilder, AdsConnection, and AdsDataAdapter components. Click OK when you are done.

Performing Basic Tasks with ADS and ADO.NET

This section describes some of the more common tasks that you can perform with the Advantage .NET Data Provider. These include connecting to a data dictionary, executing a query, using a parameterized query, retrieving and editing data, and executing a stored procedure.

Connecting to Data

You connect to a data dictionary or a directory in which free tables are located using an AdsConnection object found in the Advantage.Data.Provider namespace. At a minimum, you must provide the AdsConnection object with sufficient information to locate your data and configure how the data should be accessed. This is done using the ConnectionString property. This property accepts name/value pairs using the parameters listed in Table 18-1. If you use more than one name/value pair, separate them with semicolons.

Parameter	Description
CharType	Set to the character set type for DBF files. Valid values are ANSI and OEM. The default value is ANSI.
CommandTimeout	The number of seconds after which Advantage will cancel a long running query. The default is 30 seconds. (This setting was used in Chapter 9, *Using Notifications*, to permit a query that waits indefinitely for a notification)
CommType	The communication protocol to use to connect to ADS. Under Windows and Linux, the default is UDP_IP. For Novell Netware, the default is IPX. To use TCP/IP, set CommType to TCP_IP.
Compression	Set to ALWAYS, INTERNET, NEVER, or empty. If left empty (the default), the ADS.INI file will control the compression setting. This parameter is not used by ALS.
Connection Lifetime	The number of seconds after which a connection will be destroyed after being returned to the connection pool. The default is 0.
Connection Reset	When set to True, AdsResetConnection is called each time the connection is returned to the connection pool, there by closing all open tables, indexes, and queries, unloading loaded stored procedures, and rolling back incomplete transactions. The default value is True.
Data Source	The path to your free tables or data dictionary. If you are using a data dictionary, you must include the data dictionary filename in this path. It is recommended that this path be a UNC path. Data Source is a required parameter.
DbfsUseNulls	Set to TRUE to return empty fields from DBF files as NULL values. If set to FALSE, empty fields are returned as empty data values. The default is FALSE.

EncryptionPassword	Set to an optional password to use for accessing encrypted free tables. This parameter is ignored for data dictionary connections.
Enlist	Set to TRUE to enlist the connection in the thread's current transaction context (created by TransactionScope) when the connection is opened. The default is TRUE. This property is only applicable if you are using the .NET Framework version 2.0 or later.
FilterOptions	When FilterOptions is set to IGNORE_WHEN_COUNTING, the value returned by the AdsExtendedReader's GetRecordCount method may not reflect the number of records in a currently applied filter, but will do so when FilterOptions is set to REPECT_WHEN_COUNTING. The default is IGNORE_WHEN_COUNTING.
IncrementUserCount	Set to TRUE to increment the user count when the connection is made. Set to FALSE to make a connection without incrementing the user count. The default is FALSE.
Initial Catalog	Optional name of a data dictionary if the data dictionary is not specified in the Data Source parameter.
LockMode	Set to PROPRIETARY or COMPATIBLE to define the locking mechanism used for DBF tables. Use COMPATIBLE when your connection must share data with non-ADS applications. The default is PROPRIETARY.
Max Pool Size	The maximum number of connections to maintain in the connection pool. The default is 100.
Min Pool Size	The minimum number of connections to maintain in the connection pool. The default is 0.
Password	When connecting to a data dictionary that requires logins, set to the user's password.
Pooling	Set to TRUE to enable connection pooling. Set to FALSE to disable it. The default is TRUE.
ReadOnly	Set to TRUE to open tables readonly. Set to FALSE to open tables as editable (read-write). This setting applies to all CommandType values. The default is FALSE.
SecurityMode	Set to CHECKRIGHTS to observe the user's network access rights before opening files. Set to IGNORERIGHTS to access files regardless of the user's network rights. The default is CHECKRIGHTS. This property applies only to free table connections.
ServerType	Set to the type of ADS server you want to connect to. Use LOCAL, REMOTE, or AIS (Internet). To attempt to connect to two or more types, separate the server types using a vertical bar (|). This is demonstrated in the ConnectionString

	shown later in this chapter.
Shared	Set to TRUE to open tables shared. Set to FALSE to open tables exclusively. This setting only applies to CommandType.TableDirect. The default is TRUE.
ShowDeleted	Set to TRUE to include deleted records in DBF files. Set to FALSE to suppress deleted records. The default is FALSE.
StoredProcedure Connection	Set to TRUE if connecting from within a stored procedure. When set to TRUE, the connection does not increment the user count. The default is FALSE.
TableType	Set to ADT, VFP, CDX, or NTX to define the default table type. The default is ADT. This parameter is ignored for data dictionary connections.
TrimTrailingSpaces	Set to TRUE to trim trailing spaces from character fields. Set to FALSE to preserve trailing spaces. The default is FALSE.
User ID	If connecting to a data dictionary that requires logins, set to the user's user name.

UnicodeCollation	Provides the collation to use for Unicode fields, comparisons, etc. For example: UnicodeCollation=de_DE. To retrieve all of the supported names:
	`SELECT x.Name FROM`
	`(execute procedure sp_getcollations(NULL) ) x`
	`WHERE x.UnicodeLocale IS NULL;`

Table 18-1: The Advantage Data Provider Connection String Parameters

Note: The parameter values for CharType, LockMode, SecurityMode, ServerType, and TableType parameters also have longer name versions. For example the value ADS_ANSI can be used instead of ANSI. The longer names are recognized for compatibility with OLE DB (ADO) connection strings. You can find the longer versions of these values in Table 22-1 of Chapter 22, Advantage and MDAC, OLE DB, ADO, and Visual Basic, or in the Advantage help.

With the Advantage .NET Data Provider, the connection string property values can be enclosed in either single quotes or double quotes, if necessary. For example, if the password contains a semicolon (the connection string parameter delimiter), it would be necessary to enclose it in single quotes or double quotes.

For any of the optional connection string parameters that you fail to provide, the Advantage .NET Data Provider will automatically employ the default parameters.

Because the AdsConnection object that is used by this project must be used by a number of methods, the AdsConnection variable and several other variables that must be repeatedly referenced are declared public members of the Form class. The following is this declaration:

```
public AdsConnection connection;
public AdsCommand command;
```

```
public AdsCommand paramCommand;
public AdsDataReader dataReader;
```

The data source location of the data dictionary is also declared as a constant member of this class. This constant refers to a share named "share," on a server named "server," as shown in the following declaration:

```
private const String DataPath = "\\\\server\\share\\" +
   "adsbook\\DemoDictionary.add;";
```

This connection, named AdsConnection, is created, configured, and opened from the InitializeDataComponents method of the form, along with several other objects. InitializeDataComponents is called from the Form's constructor. The relevant portion of this custom private method is shown in the following code:

```
private void InitializeDataComponents() {
  connection = new AdsConnection();
  connection.ConnectionString = "Data Source=" + DataPath +
    ";user ID=adsuser;password=password;" +
    "ServerType= LOCAL | REMOTE;TrimTrailingSpaces=True";
  connection.Open();
  command = new AdsCommand();
  command = connection.CreateCommand();
  //additional statements follow
```

Note: If you have difficulty connecting, it might be because you have other client applications, such as the Advantage Data Architect, connected using a local connection. Ensure that all clients on the same machine use the same type of connection.

Executing a Query

You execute a query that returns a result set using an AdsCommand. There are numerous overloaded methods for doing this. The following code segment demonstrates one of these, where a query string and an open connection are passed as parameters to an AdsDataAdapter's constructor. Within this constructor, the query string is assigned to an internally created AdsCommand object that is associated with the SelectCommand property of the AdsDataAdapter, which performs the query execution. The Fill method is then invoked on this AdsDataAdapter, which causes the result set to be loaded into a DataTable of a DataSet.

This DataTable is then used to display the resulting data in a DataGridView, as shown in Figure 18-4. The following code demonstrates the execution of a query entered by the user into the TextBox named selectText. This method is associated with the Execute SELECT button which is shown earlier in Figure 18-3:

```
private void executeSELECTBtn_Click(object sender,
  System.EventArgs e) {
  IDataAdapter adapter ;
  adapter = new AdsDataAdapter(selectText.Text, connection);
  DataSet ds = new DataSet();
```

```
  adapter.Fill(ds);
  DataTable dt = ds.Tables[0];
  dataGridView1.DataSource = dt;
}
```

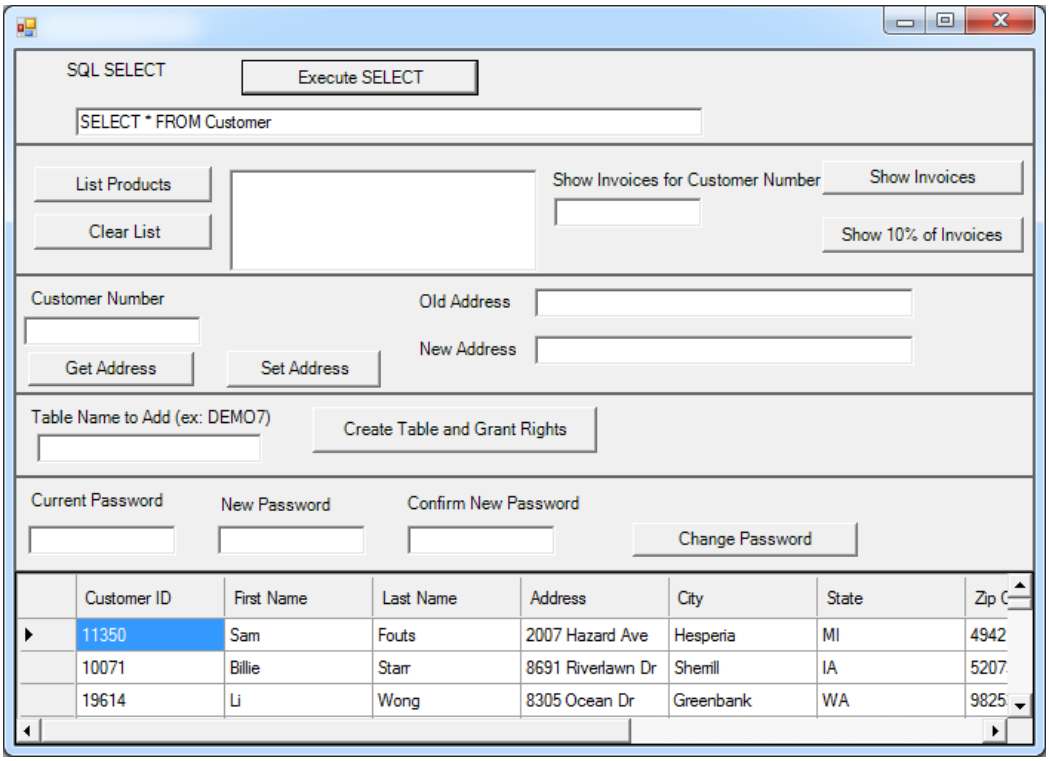

Figure 18-4: The results of a SELECT query displayed in a DataGridView

Notice that the AdsDataAdapter that is created is assigned to a variable of type IDataAdapter. IDataAdapter is the interface that all data adapters implement. While we could have just as well assigned this object to a variable of type AdsDataAdapter, assigning it to an interface variable makes our code more portable, since any IDataAdapter implementing class can be assigned to this variable. This technique is used in this chapter whenever no Advantage .NET Data Provider functionality is specifically needed. When we need a feature specific to an AdsDataAdapter, our variable is of that class type.

If you need to execute a query that does not return a result set, call the ExecuteNonQuery method of an AdsCommand object. The use of an AdsCommand object to execute a query that does not return a result set is demonstrated later in this chapter.

Using a Parameterized Query

Parameterized queries are defined using an AdsCommand object. Before you can invoke a parameterized query, you must create one AdsParameter object for each of the

query's parameters. You can create an AdsParameter instance by calling the Add method of the AdsCommand object's Parameters property.

The definition of a parameterized query, including the creation of a parameter, is shown in the following code segment, which is part of the private InitializeDataComponents method shown earlier:

```
paramCommand = new AdsCommand("SELECT * FROM INVOICE " +
  "WHERE [Customer ID] = ?", connection);
paramCommand.Parameters.Add(1, typeof(Int32));
```

Before you can execute an AdsCommand that contains a parameterized query, you must bind data to each of its parameters. This is shown in the following method, which is called by the Show Invoices button shown in Figure 18-3:

```
private void doParamQuery_Click(object sender,
  System.EventArgs e) {
  IDataAdapter dataAdapter;
  DataSet ds = new DataSet();
  DataTable dt;
  if (paramText.Text.Equals("")) {
    MessageBox.Show(this,
      "You must supply a customer ID");
    return;
  }
  paramCommand.Parameters[0].Value =
    Int32.Parse(paramText.Text);
  dataAdapter = new AdsDataAdapter(paramCommand);
  dataAdapter.Fill(ds);
  dt = ds.Tables[0];
  if (dt.Rows.Count == 0)
  {
    MessageBox.Show(this,
      "No invoices for customer ID");
    return;
  }
  dataGridView1.DataSource = dt;
}
```

As you can see from this code, after verifying that a value has been entered into the Customer ID field, the entered data is assigned to the Value property of the AdsParameter. The AdsCommand object that holds the parameter is passed as an argument to an AdsDataAdapter, which then executes the query and assigns the result set to a DataTable (using the Fill method). Note that it was not necessary to pass a connection object to the AdsDataAdapter constructor, since the AdsCommand object itself was constructed based on a connection.

This is actually a classic example of how parameterized queries are used. Specifically, the query text is defined only once, but can be executed repeatedly. And by changing only the value of the parameter, a different result set can be returned upon each execution.

Reading and Writing Data

ADO.NET supports three mechanisms for reading data obtained through a SQL SELECT query. Which mechanism you use depends on how much data you are reading, and what you plan to do with it.

In the simplest case, if you are reading a single value from a table, you can call the ExecuteScalar method of an AdsCommand.

The remaining two mechanisms permit you to read multiple fields and multiple records. If you need to be able to refer to multiple rows of data at the same time, you will likely load the data through an AdsDataAdapter into a DataTable. This approach is demonstrated earlier in this chapter in the section on executing a query. In that example, a DataTable is populated and its contents displayed in a DataGridView.

The third mechanism is to use a class that implements the IDataReader interface, and the Advantage .NET Data Provider offers two classes that implement that interface.

The AdsDataReader is a typical ADO.NET data reader implementation. Like other data readers, it provides a forward-only, readonly cursor to a result set. After obtaining an AdsDataReader by calling the ExecuteReader method of an AdsCommand, you call the Read method. If Read returns the value True, the AdsDataReader points to the first record in the result set, after which you call any of the available Get methods, such as GetString, GetBoolean, or GetDate, to read data from that record.

If there are additional records in the result set, calling Read advances the data reader to the next record. Read returns False when there are no remaining records in the result set referred to by the data reader.

The following code segment demonstrates how to read data using an AdsDataReader. This code is associated with the event handler assigned to the Get Address button shown in Figure 18-3:

```
private void getAddressBtn_Click(object sender,
  System.EventArgs e)
{
  AdsCommand getCustCommand;
  AdsDataReader dataReader;
  String custNo;

  custNo = custNoText.Text;

  if (custNo.Equals(""))
  {
    MessageBox.Show(this, "You must supply a customer ID");
    return;
  }

  getCustCommand = new AdsCommand(
    "SELECT * FROM CUSTOMER WHERE [Customer ID] = ?",
    connection);
  getCustCommand.Parameters.Add(1, typeof(Int32));
  getCustCommand.Parameters[0].Value =
  Int32.Parse(custNo);
```

```
  dataReader = getCustCommand.ExecuteReader();
  try
  {
    if (dataReader.Read())
    {
      oldAddressText.Text =
        dataReader.GetString
      (dataReader.GetOrdinal("Address"));
    }
    else
    {
      MessageBox.Show(this, "Customer " + custNo +
        " not found");
    }
  }
  finally
  {
    dataReader.Close();
  }
}
```

Most ADO.NET developers write data using SQL queries. These may be invoked directly through an IDbCommand object, or they can be invoked through a properly configured IDbDataAdapter.

As mentioned earlier in this section, Advantage developers have an additional tool for writing data, the AdsExtendedReader. The AdsExtendedReader supports both read and write operations on a live cursor returned by a SQL SELECT statement or from a table opened directly. (Opening a table directly, by setting the AdsCommand's CommandType to TableDirect, permits you to open a table exclusively, so long as the shared=false name/value pair appears in the command's connection string.)

Since the AdsExtendedReader is unique to Advantage, and offers editing capabilities that are unique in the world of ADO.NET, the following example demonstrates how to write to a table using an AdsExtendedReader. For information on writing data using the Update method of a data adapter, refer to the .NET framework documentation.

Actually, using an AdsExtendedReader is almost as simple as using an AdsDataReader. The primary difference is that the AdsExtendedReader has many more methods than the typical data reader. After calling ExecuteExtendedReader, which executes the associated SQL SELECT statement, you can insert records, delete records, read fields, write to fields, set a range, apply a filter, perform a seek, empty the table, or run almost any other operation that you would expect from a server-side cursor.

The following code demonstrates a simple write operation using an AdsExtendedReader. Like the preceding example, an AdsExtendedReader is used to retrieve a single record from the Customer table. Once retrieved, the selected customer's address is changed, and the update is written to the underlying table. This code is associated with the event handler assigned to the Set Address button shown in Figure 18-3.

```
private void setAddressBtn_Click(object sender,
  System.EventArgs e)
{
  AdsCommand getCustCommand;
  AdsExtendedReader extendedReader;
  String custNo;

  custNo = custNoText.Text;

  if (custNo.Equals(""))
  {
    MessageBox.Show(this, "You must supply a customer ID");
    return;
  }

  getCustCommand = new AdsCommand(
      "SELECT * FROM CUSTOMER WHERE [Customer ID] = ?",
      connection);
  getCustCommand.Parameters.Add(1, typeof(Int32));
  getCustCommand.Parameters[0].Value =
    Int32.Parse(custNo);

  extendedReader = getCustCommand.ExecuteExtendedReader();
  try
  {
    if (extendedReader.Read())
    {
      extendedReader.SetString(
      extendedReader.GetOrdinal("Address"),
        newAddressText.Text);
      extendedReader.WriteRecord();
    }
    else
    {
      MessageBox.Show(this, "Customer " + custNo +
        " not found");
    }
  }
  finally
  {
    extendedReader.Close();
  }
}
```

Note: Whether you use an AdsDataReader or an AdsExtendedReader, it is very important to call the data reader's Close method when you are done with it. You cannot use the AdsCommand object from which you created the data reader for any other queries until the data reader it returned has been closed. This is why the preceding examples have included the call to Close in a finally clause.

Calling a Stored Procedure

Calling a stored procedure is no different than executing any other query. You can define a SQL EXECUTE PROCEDURE statement and assign it to the CommandText property of an AdsCommand object. Alternatively, you can assign the name of the stored procedure to the CommandText property of the AdsCommand object, and then set the CommandType property to CommandType.StoredProcedure. And in Advantage 10, you can include a stored procedure in the FROM clause of a select query, as described in Chapter 12, *Introduction to Using Advantage SQL*.

You create an AdsCommand object explicitly or permit an AdsDataAdapter or an AdsConnection to create one for you. If the stored procedure returns a result set, you call the AdsCommand ExecuteReader method, or the Fill method of the AdsDataAdapter that refers to the AdsCommand in its SelectCommand property. If the stored procedure returns a single value, you can also call the ExecuteScalar method of an AdsCommand. Finally, if the stored procedure does not return a result set, execute it by calling the ExecuteNonQuery method of the AdsCommand.

Invoking a stored procedure that takes one input parameter is demonstrated by the following code associated with the Show 10% of Invoices button (shown in Figure 18-3). The stored procedure referenced in this code is the SQL stored procedure you created in Chapter 7, *Creating Stored Procedures*, called Get10PercentSQL.

```
private void callStoredProc_Click_1(object sender,
  System.EventArgs e) {
  AdsCommand storedProcCommand;
  IDataAdapter dataAdapter;
  DataSet ds = new DataSet();
  DataTable dt;
  if (paramText.Text.Equals(""))
  {
    MessageBox.Show(this, "You must supply a customer ID");
    return;
  }
  storedProcCommand = new AdsCommand("Get10PercentSQL",
    connection);
  storedProcCommand.CommandType =
    CommandType.StoredProcedure;
  storedProcCommand.Parameters.Add(1,
    System.Data.DbType.Int32);
  storedProcCommand.Parameters[0].Value =
    Int32.Parse(paramText.Text);
  dataAdapter = new AdsDataAdapter(storedProcCommand);
  dataAdapter.Fill(ds);
  dt = ds.Tables[0];
  if (dt.Rows.Count == 0)
  {
    MessageBox.Show(this, "No invoices for customer ID");
    return;
  }
  dataGrid1.DataSource = dt;
}
```

Navigational Actions with Advantage and ADO.NET

Within the ADO.NET framework itself, most of the navigational features that you can access with other data access mechanisms, such as ADO (ActiveX Data Objects), Visual FoxPro, and Delphi, are implemented in the ADO.NET DataSet, DataTable, or DataView classes. For example, indexing, sorting, filtering, and seeking are all operations that are performed on a DataTable's cached records using a DataView. In other words, these operations do not involve Advantage, other than using Advantage as the original source of the data that is manipulated in memory.

There is, in fact, only one ADO.NET navigational operation that involves Advantage—scanning. Specifically, using an AdsDataReader (or an AdsExtendedReader, which as you learned earlier, also supports filters, ranges, and seeks), you can perform a record-by-record navigation of data. This operation is demonstrated in the following section.

Scanning a Result Set

Scanning is the process of sequentially reading every record in a result, or every record in the filtered view of the result set if a filter is active. In ADO.NET, scanning is performed using an AdsDataReader or AdsExtendedReader, which you obtain from an AdsCommand object that contains either a SQL SELECT statement, a table name, or a stored procedure call.

Tip: If you are using ADS, and you must scan a large number of records, consider implementing the operation using a stored procedure as described in Chapter 7, Creating Stored Procedures. Scanning from a stored procedure installed on the same machine on which the data resides requires no network resources.

The following code demonstrates scanning using an AdsDataReader. This code is associated with the List Products button shown in Figure 18-3:

```
private void listProductsBtn_Click(object sender,
    System.EventArgs e)
{
  command.CommandText = "SELECT * FROM PRODUCTS";
  dataReader = command.ExecuteReader();
  try
  {
    while (dataReader.Read())
    {
      productList.Items.Add(dataReader.GetString(0) + "  " +
        dataReader.GetString(1));
    }
  }
  finally
  {
    dataReader.Close();
```

```
  }
}
```

Administrative Operations with ADS and ADO.NET

While Advantage requires little in the way of periodic maintenance to keep it running smoothly, many applications need to provide administrative functionality related to the management of users, groups, and objects.

This section is designed to provide you with insight into exposing administrative functions in your client applications. Two related, yet different, operations are demonstrated here. In the first, a new table is added to the database and all groups are granted access rights to it. This operation requires that you establish an administrative connection (or a connection from a user account with GRANT privileges). The second operation involves permitting individual users to modify their own passwords. Especially in the security-conscious world of modern database management, this feature is often considered an essential step to protecting data.

Creating a Table and Granting Rights to It

The CS_ADONET project permits a user to enter the name of a table that will be created in the data dictionary, after which, all groups will be granted rights to the table. This operation is demonstrated in the following method, which is associated with the Create Table and Grant Rights button shown in Figure 18-3:

```
private void addTableBtn_Click(object sender,
  System.EventArgs e)
  {
  AdsConnection adminConnection;
  IDbCommand adminCommand;
  AdsDataAdapter adminAdapter;
  DataSet ds;
  DataTable dt;
  DataRow dr;
  String tabName;

  tabName = tableNameText.Text;
  if (tabName.Equals(""))
  {
    MessageBox.Show(this,
      "Please enter the name of the table to create");
    return;
  }
  //Check for SQL injections hack
  if ((tabName.IndexOf(";") >= 0))
  {
    MessageBox.Show(this,
      "Table name may not contain a semicolon");
   return;
  }
  try
  {
    adminConnection = new AdsConnection(
```

```
      "Data Source=" + DataPath + ";user ID=adssys;" +
      "password=password;" +
      "ServerType=ADS_LOCAL_SERVER | ADS_REMOTE_SERVER;" +
      "TrimTrailingSpaces=True");
    adminConnection.Open();
    adminAdapter = new AdsDataAdapter("SELECT Name FROM " +
      "system.tables " +
      "WHERE UCase(Name) = '" + tabName.ToUpper() + "'",
      adminConnection);
    ds = new DataSet();
    adminAdapter.Fill(ds);
    dt = ds.Tables[0];
    if (dt.Rows.Count != 0)
    {
      MessageBox.Show(this,
        "This table already exists. Cannot create");
      return;
     }
    adminCommand = new AdsCommand("CREATE TABLE [" +
      tabName + "] " +
      "([Full Name] CHAR(30), [Date of Birth] DATE," +
      "[Credit Limit] MONEY, Active LOGICAL)",
       adminConnection);
    adminCommand.ExecuteNonQuery();
    adminAdapter = new AdsDataAdapter("SELECT Name FROM " +
      "system.usergroups WHERE Name NOT LIKE 'DB:%'", adminConnection);
    ds = new DataSet();
    adminAdapter.Fill(ds);
    dt = ds.Tables[0];
    if (dt.Rows.Count == 0)
    {
      MessageBox.Show(this, "No groups to grant rights to");
      return;
    }
    adminCommand = adminConnection.CreateCommand();
    for (int i=0; i <= dt.Rows.Count - 1 ; i++)
    {
      dr = dt.Rows[i];
      adminCommand.CommandText = "GRANT ALL ON " +
        tabName + " TO \"" + dr[0].ToString() + "\"";
      adminCommand.ExecuteNonQuery();
    }
  }
  catch (System.Exception ex)
  {
    Console.WriteLine("Exception", ex);
    throw(ex);
  }
  MessageBox.Show(this,
    "The " + tabName + " table has been " +
    "created, with rights granted to all groups");
  return;
}
```

This method begins by verifying that the table name does not include a semicolon, which could be used to convert the subsequent GRANT SQL statement into a SQL script. Since this value represents a table name, a parameterized query is not an option.

Next, this code verifies that the table does not already exist in the data dictionary. Once that is done, a new connection is created using the data dictionary administrative account. This connection is then used to call CREATE TABLE to create the table, and then to call GRANT for each non-default group returned in the system.usergroups table.

Note: The administrative user name and passwords are represented by string literals in this code segment. This was done for convenience, but in a real application, you would either ask for this information from the user or you would obfuscate this information so that it could not be retrieved.

Changing a User Password

A user can change the password on their own connection, if you permit this. In most cases, only when every user has a distinct user name would you expose this functionality in a client application. When multiple users share a user name, this operation is usually reserved for an application administrator.

The following method, associated with the Change Password button (shown in Figure 18-3), demonstrates how you can permit a user to change their password from a client application:

```
private void changePasswordBtn_Click(object sender,
  System.EventArgs e)
{
  IDataReader dataReader;
  String userName;
  String oldPass, newPass, confirmPass;
  oldPass = oldPassText.Text;
  if (oldPass.Equals(""))
  {
   MessageBox.Show(this,
    "Please enter your current password");
   return;
  }
  newPass = newPassText.Text;
  if (newPass.Equals(""))
  {
   MessageBox.Show(this,
    "Please enter your new password");
   return;
  }
  confirmPass = confirmPassText.Text;
  if (confirmPass.Equals(""))
  {
   MessageBox.Show(this,
    "Please confirm your new password");
   return;
  }
  if (! newPass.Equals(confirmPass))
  {
   MessageBox.Show(this,
    "New passwords do not match");
```

```
   return;
  }
  if ((newPass.IndexOf(";") >= 0))
  {
   MessageBox.Show(this,
    "Password may not contain a semicolon");
   return;
  }
  //Get user name
  command = connection.CreateCommand();
  command.CommandText = "SELECT USER() FROM system.iota";
  dataReader = command.ExecuteReader();
  dataReader.Read();
  userName = dataReader.GetString(0);
  dataReader.Close();
  //Verify current password
  if (! CheckPassword(userName, oldPass))
  {
   MessageBox.Show("Cannot validate your current password. "
     +"Cannot change password");
   return;
  }

  try
  {
    command.CommandText =  "EXECUTE PROCEDURE " +
      "sp_ModifyUserProperty('" +
      userName + "', 'USER_PASSWORD', '" + newPass + "')";
    command.ExecuteNonQuery();
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
    return;
  }
  MessageBox.Show("Password successfully changed. " +
    "New password will be valid next time you connect");
}
```

A number of interesting tricks are used in this code. First, the user name is obtained by requesting the USER scalar function from the system.iota table. USER returns the user name on the connection through which the query is executed. Next, the user is asked for their current password, and the user name and password is used to attempt a new connection, which if successful confirms that the user is valid. This password validation is performed using the custom CheckPassword method. The following is the implementation of this method:

```
private Boolean CheckPassword(String uName, String pass)
{
  //Verify the current password
  AdsConnection tempConnection;
  tempConnection = new AdsConnection(
    "Data Source=" + DataPath + ";user ID=" + uName + ";" +
    "password=" + pass + ";" +
    "ServerType=ADS_LOCAL_SERVER | ADS_REMOTE_SERVER;" +
    "TrimTrailingSpaces=True");
```

```
  try
  {
    tempConnection.Open();
    tempConnection.Close();
    return true;
  }
  catch (Exception)
  {
    return false;
  }
} //CheckPassword
```

Finally, the user is asked for their new password twice (for confirmation). If all is well, the sp_ModifyUserProperty stored procedure is called to change the user's password. This password will be valid once the user terminates all connections on this user account.

*Note: If you run this code, and change the password of the adsuser account, you should use the Advantage Data Architect to change the password back to **password**. Otherwise, you will not be able to run this project again since the password is hard-coded into the connection string.*