Delve into the details of using ADO.Net and Informix® Dynamic Server beyond the basic connection string. This article covers stored procedures and parameterized SQL. Then it presents a model (with code) for generating your own strongly typed DataSets.

Introduction

With its massive .NET framework, Microsoft® introduced a new data access technology called ADO.NET. In this article, we will examine how to use the ADO.NET driver for Informix that is included with the IBM Client SDK version 2.90. The sample code included is written in C#.

Laying the groundwork

Before using the ADO.NET driver, you should make sure it is installed and working properly. The current version of the driver is installed with the Informix Client Software Developer's Kit (SDK) 2.90. Unlike the previous 2.81 version, this SDK installation includes the ADO.NET driver by default. As you might expect, the client machine must have the .NET framework installed in order to use the driver. The SDK install program will warn you about this, too. It does not seem to actually look for the .NET framework, it just warns you that it needs to already be installed. If you have the 2.81 SDK already installed, it is best to uninstall it first. The two versions do not co-exist well. Be aware that the 2.90 ADO.NET driver incorrectly reports itself as version 2.81 when you add it into your Visual Studio Projects.

The 2.90 version is a significant upgrade over the 2.81 version. It includes a new IfxDataAdapter wizard, IPv6 support, and new classes for Informix data types (IfxDateTime, IfxDecimal, IfxBlob, and IfxClob). The documentation is more complete with twice the amount of material.

Important: The IBM Informix ADO.Net driver is not self-contained in the IBM.Data.Informix.dll file that gets installed in the /bin directory of your installation. Apparently, it uses some of the other client code installed by the SDK. This means that you must install the Informix Client SDK on any machines that will use the ADO.Net driver. You cannot just include the IBM.Data.Informix.dll in your distribution. This could be a serious limitation for some applications. You also need to go through the SDK setup (SetNet32) to define your Informix data sources.
Before using the ADO.NET driver to connect, you must also run a stored procedure called cdotnet.sql. It is located in the /etc directory of your SDK installation. This is similar to the process of setting up the OLEDB driver, though the procedure is much shorter. This process is documented in the User's Guide. (See the Related topics section below.)

After installation, check your driver and make sure you get a connection. To use the ADO.NET driver in your Visual Studio project, make sure you add a reference to the IBM.Data.Informix.dll found in the /bin directory of your client SDK installation. The proper using statement is:

```csharp
using IBM.Data.Informix;
```

Here is a simple method that demonstrates how to get a connection to the database:

### Listing 1. Connecting to an Informix database

```csharp
public void MakeConnection() {
    string ConnectionString = "Host=" + HOST + "; " +
                            "Service=" + SERVICENUM + "; " +
                            "Server=" + SERVER + "; " +
                            "Database=" + DATABASE + "; " +
                            "User Id=" + USER + "; " +
                            "Password=" + PASSWORD + "; " +
    //Can add other DB parameters here like DELIMIDENT, DB_LOCALE etc
    IfxConnection conn = new IfxConnection();
    conn.ConnectionString = ConnectionString;
    try {
        conn.Open();
        Console.WriteLine("Made connection!");
        Console.ReadLine();
    } catch (IfxException ex) {
        Console.WriteLine("Problem with connection attempt: " + ex.Message);
    }
}
```

The sample code includes a `BasicConnection` class for this functionality. As you can see, the `ConnectionString` is just a semicolon-separated list of parameters for the connection. The `open()` method opens the connection to the database and throws an `IfxException` if the connection fails. The `IfxException.Message` property usually gives a reasonable amount of detail on the reason for the failure.

### Basic commands

Once you have a connection, you can begin to execute commands against the database. To do this, use an `IfxCommand` object. The constructor for an `IfxCommand` takes a string (the SQL command text) and an `IfxConnection`. The `IfxCommand` object has a series of `Execute` methods to execute the command against the database. To clean up, use the `IfxConnection.Close()` method. Here is an example of executing a simple command that doesn't return a set of results. It could be an insert, update, or delete.
Listing 2. Executing an insert, update or delete

```csharp
IfxCommand cmd;
cmd = new IfxCommand("insert into test values (1, 2, 'ABC')", conn);
try {
    int rows = cmd.ExecuteNonQuery();
} catch (IfxException ex) {
    Console.WriteLine("Error " + ex.Message);
}
```

ExecuteNonQuery returns, as an integer, the number of rows affected by the command. You can also build parameterized statements and queries, which we will also examine below. Notice the CommandTimeout property of the IfxCommand. The default timeout is 30 seconds, although it is undocumented. Unless you change this property, a command that runs for over 30 seconds will time out and throw an exception.

The next example is executing a select statement and working with the set of results returned by the database server. For a fast, forward-only cursor through the results, use an IfxDataReader returned by the ExecuteReader method. However, you can only have one open IfxDataReader per IfxConnection. (This is an ADO.NET limitation, not a specific limitation of the Informix ADO.NET driver.)

Listing 3. Iterating through an IfxDataReader

```csharp
IfxCommand cmd = new IfxCommand("select * from test", bconn.conn);
try {
    IfxDataReader dr = cmd.ExecuteReader();
    while (dr.Read()) {
        int a = dr.GetInt32(0);
        int b = Convert.ToInt32(dr["b"]);
        string c = (String)dr[2];
    }
    dr.Close();
} catch (IfxException ex) {
    Console.WriteLine("Error " + ex.Message);
}
```

Each column is retrieved as a generic Object type. As the code demonstrates, there are several ways to convert the column Objects into the correct datatypes. You can use the GetXxx methods of the IfxDataReader. There are methods for almost every datatype. The GetXxx methods take a column number as a parameter. You can use the indexers of the IfxDataReader to access the columns by their names. The .NET framework Convert functions will convert these Objects into the proper types, if possible. Finally, you can index into the columns by column number and cast the results directly (for some types).

This next example shows how to call a stored procedure that needs a parameter value.
Listing 4. Executing a stored procedure with a parameter

```csharp
IfxCommand cmd = new IfxCommand("test_proc", conn);
    cmd.CommandType = CommandType.StoredProcedure; //from System.Data
    cmd.Parameters.Add("in_parameter", 2); //many ways to create these
    try {
        cmd.ExecuteScalar();
    }
    catch (IfxException ifxe) {
        Console.WriteLine("Error "+ifxe.Message);
    }
```

For this `IfxCommand`, you must set the `CommandType` to the `StoredProcedure` value from the `CommandType` enum in System.Data. To create the parameter, you can use the `IfxCommand`'s `Parameters.Add` method. `IfxCommands.Parameters` is a collection, so you can add as many parameters as you need. You can create the parameters with any of the `IfxParameter()` constructors, or you can shortcut their creation as above. Note, however, that each `IfxParameter` is associated with a specific `IfxCommand`. You cannot create `IfxParameters` and then use them in multiple `IfxCommand` objects. The `ExecuteScalar()` method returns only 1 row. This example does not return anything back from the stored procedure.

To build a parameterized SQL statement that doesn't execute a stored procedure, insert question marks as place-holders in the `CommandText`. For example:

Listing 5. Parameterized query

```csharp
IfxCommand insCmd = new IfxCommand("insert into clientstest "
    + "(clientcode, clientacctname, primarycontact, primaddrcode, "
    + "initialamt,createdate) values (?,?,?,?,?,TODAY)"
    , conn);
```

Add `IfxParameter` objects to the `IfxCommand`'s `Parameters` collection in the exact order as they occur in the command text. This technique is further demonstrated in the final strongly typed DataSets in the extended example below.

Strongly typed DataSets

ADO.NET includes a specialized database object called a `DataSet`. It is an in-memory database. The `DataSet` consists of one or more `DataTable` objects (made up of `DataRow` objects). The `DataTables` can be related by primary and foreign keys. And constraints can be placed on the data. The `DataSet` is also disconnected from the actual data store. It gets filled through one or more `DataAdapters` (one per `DataTable`), and then keeps that data and any changes in memory. At a later point, the `DataAdapters` can submit changes back to the data store.

The basic `DataSet` is not strongly typed. It does not know what the real columns and rows of the database are. Columns can be indexed by name: `row["itemcode"]`. But compiler does not check these column names. Any mistake in the column name, for example, is not apparent until runtime. Also, the developer has no help remembering if the column is "itemcode" or "itemid."

A strongly typed `DataSet` addresses these problems. Instead of a generic `DataRow`, it would have, for example, an `OrderDetailDataRow` as part of an `OrderDetailDataTable`. And you could refer to columns as actual properties of the `OrderDetailDataTable` `row.ItemCode`. This way you also get
the productivity benefits of Intellisense. The table and column names also become available in the property editors to enhance designer-level tools like data binding.

So how can you build this productivity-enhancing, strongly typed *DataSet*? Will it take so much time or effort to build that you don't experience any net productivity gain? The Informix ADO.NET driver may not be as sophisticated as some other drivers. The Microsoft *SQLDataAdapter* (for SQL Server) includes a Generate DataSet wizard. The *IfxDataAdapter* doesn't have this wizard yet. However, you can build some tools to help you. You can also use some tools already built into the .NET framework. In the end, you will have a descendent of a strongly typed *DataSet* that encapsulates all of the database interaction.

The .NET framework includes an XSD compiler (*xsd.exe*) that can generate a strongly typed *DataSet* from a specially formatted .xsd file. But who wants to type in a bunch of XML? Fortunately, the *DataSet* object includes a method called *WriteXmlSchema()* . This method allows you to use a non-typed *DataSet* to create the XSD file for a strongly-typed *DataSet*. Let's look at how this works.

Here is a simple table:

**Listing 6. Clientstest table**

```sql
CREATE TABLE clientstest (  
  clientcode SERIAL not null,  
  clientacctname CHAR(60) not null,  
  primarycontact CHAR(30) not null,  
  primaddrcode CHAR(10),  
  createdate DATE,  
  initialamt DECIMAL (18,0)  
);
```

Here's the single-table *DataSet* for that table:

**Listing 7. Defining the *DataSet***

```csharp
DS = new DataSet("dsClients");

//main table definition
DataTable mainTable = new DataTable("clients");
DataColumnCollection cols = mainTable.Columns;
DataColumn column = cols.Add("clientcode", typeof(Int32));
column.AllowDBNull = false;
cols.Add("clientacctname", typeof(String)).MaxLength = 60;
cols.Add("primarycontact", typeof(String)).MaxLength = 30;
cols.Add("primaddrcode", typeof(String)).MaxLength = 10;
cols.Add("initialamt", typeof(Decimal));
cols.Add("createdate", typeof(System.DateTime));

//primary key
mainTable.PrimaryKey = new DataColumn[] {cols["clientcode"]};

//add table to *DataSet*
DS.Tables.Add(mainTable);

//Write schema to file
DS.WriteXmlSchema("dsClients.xsd");
```
In this definition, you set the types and the constraints on the data. You also set the names for the columns. They do not have to match the database's column names. Look in the code files in the Download section of this article to see the resulting dsClients.xsd file.

To make it easier to generate the XSD file (and re-generate it, after changes), build a framework for these DataSet Builders. (All the code required for this is included below.) Since you want the framework to determine which Builders to build, use reflection to dynamically determine if something is a DataSetBuilder. Start by writing the IBuildable interface. It defines the properties and methods that our DataSetBuilders must implement.

Listing 8. IBuildable interface

```csharp
public interface IBuildable {
    string FileName {get; set;}
    string FilePath {get; set;}
    Logger Log {get; set;}
    DataSet DS {get; set;}

    void BuildXSD();
    void CompileXSD(string outputDirectory);
}
```

The code from listing 7 (the definition of the DataSet) is basically the BuildXSD() method. Create an abstract parent class called DataSetBuilder. BuildXSD is the abstract method that must be overridden in each concrete descendent. The CompileXSD method is the same for each DataSetBuilder, so it resides in the DataSetBuilder. Here is the CompileXSD() method from that abstract class:

Listing 9. CompileXSD() method

```csharp
log.LogStatus("Compiling "+filename);
ProcessStartInfo processinfo = new ProcessStartInfo();
processinfo.FileName = @"C:\Program Files\Microsoft Visual Studio .NET 2003" +@"SDK\v1.1\Bin\xsd.exe";
processinfo.Arguments = FilePath+FileName+" /dataset /namespace:" +ds.Namespace+" /out:"+outputDirectory+" /ns:"+outputNamespace;  //doesn't work
processinfo.UseShellExecute = false;
processinfo.RedirectStandardInput = true;
processinfo.RedirectStandardError = true;
processinfo.CreateNoWindow = true; //doesn't work
processinfo.RedirectStandardInput = true;
processinfo.RedirectStandardOutput = true;
processinfo.RedirectStandardError = true;
processinfo.CreateNoWindow = true; //doesn't work
processinfo.UseShellExecute = false;

Process compiler = Process.Start(processinfo);
log.LogStatus("Output:
"+compiler.StandardOutput.ReadToEnd());
log.LogStatus("Error:
"+compiler.StandardError.ReadToEnd());
compiler.WaitForExit();
```

This method uses the Process and ProcessStartInfo classes from System.Diagnostics in the .NET framework to execute the XSD compiler. This example code uses the free and relatively simple .NET Logging Framework from the ObjectGuy (see Related topics).

Because the DataSetBuilder classes all implement the IBuildable interface, we can use reflection to look through the assembly and build all the DataSet classes from the DataSetBuilders. This is
what the DataLibraryBuilder class does. For example, the ClientsBuilder gets compiled to a dsClients class.

The generated dsClients class is the strongly-typed dataset. From the 42-line ClientsBuilder, we now have almost 500 lines of strongly typed code. Look at this generated code. It contains a clientsDataTable. The clientsDataTable has properties for each column. There are also methods like: NewclientsRow(), IsinitialamtNull(), and FindByclientcode(int clientcode). These will be quite useful when using this class.

To encapsulate the Informix database access into the strongly typed dataset, inherit from dsClients. This is the clients class that you can use in your application. Inheritance provides some protection from changes in the schema of the DataSet. If the schema changes, you can just regenerate the dsClients class. The clients class remains unchanged (though you may need to make changes there, too). In the clients class, add an IfxDataAdapter for each DataTable (just one in this case). For each IfxDataAdapter, define the SQL text and parameters for the select, insert, update, and delete commands. You can then override the Fill and Update methods to initialize, fill, and update all the IfxDataAdapters. Look at the Insert Command as an example:

**Listing 10. InsertCommand for the IfxDataAdapter**

```csharp
IfxCommand insCmd = new IfxCommand("insert into clientstest "+"(clientcode, clientacctname, primarycontact, primaddrcode, "+
"initialamt,createdate) values (?,?,?,?,TODAY)", conn);
insCmd.Parameters.Add("clientacctname", IfxType.Char, 60,
"clientacctname");
insCmd.Parameters.Add("primarycontact", IfxType.Char, 30,
"primarycontact");
insCmd.Parameters.Add("primaddrcode", IfxType.Char, 10,
"primaddrcode");
insCmd.Parameters.Add("initialamt", IfxType.Decimal, 16,
"initialamt");
daclients.InsertCommand = insCmd;
```

The IfxDataAdapter has the following command properties: SelectCommand, InsertCommand, DeleteCommand, and UpdateCommand. When the IfxDataAdapter executes the Fill() method, it uses the SelectCommand to query the database. When the Update() method is called, the IfxDataAdapter uses a combination of the Insert, Update, and Delete commands to conform the database to the in-memory version of the table. The IfxDataAdapter decides which rows and columns need to be changed; the developer does not have to code to track the changes to the data.

Notice that zero is inserted into the serial value, as is usual for the Informix serial type. But how can you get the database-generated value back into your disconnected DataSet? You have to hook into the RowUpdated event of the dsclients IfxDataAdapter. In that event handler, the code looks for any inserts. For an insert, it executes a dbinfo command to retrieve the just-created serial value. It puts that value into the clientcode column for that DataRow. Here's the event handler code for this Informix-specific trick:
Listing 11. Retrieving the generated serial value

```csharp
private void daclients_RowUpdated(object sender, 
    IfxRowUpdatedEventArgs e) {
    //For INSERTs only, gets the serial id and 
    //inserts into the clientcode
    if (e.StatementType == StatementType.Insert) {
        IfxCommand getSerial = new IfxCommand(
            "select dbinfo('sqlca.sqlerrd1') from systables "
            +"where tabid = 1",
            daclients.InsertCommand.Connection);
        e.Row["clientcode"] = (int)getSerial.ExecuteScalar();
    }
}
```

Results

You now have a fully encapsulated business object that handles its own database interaction. What can you do now? Since DataSets derive from System.Component, you can add your strongly typed DataSet objects (for example, Clients) onto your Visual Studio Toolbox. Then you can drag it out onto any WinForm or WebForm design view. Set the property for the Connection. In your code, execute the object's Fill() method (perhaps in a FormLoad event). That fills the object with all the data for each DataTable. In the Designer view, you can also databind by setting the DataSource (and perhaps theDataMember property) for the visual control or grid.

The LibraryConsoleTest program in the sample solution demonstrates how the strongly typed DataSet works. You can now write something like this: `Console.WriteLine(client.clientcode + " " + client.clientacctname + " " + client.createdate);` instead of this: `Console.WriteLine(ds.Tables["clients"].Rows["clientcode"] + " " + ds.Tables["clients"].Rows["clientacctname"] + " " + ds.Tables["clients"].Rows["createdate"] );` The LibraryConsoleTest adds a new client and retrieves the generated serial number. It deletes a client after using the Findbyclientcode() method to select the proper row. It also updates one column in a particular row. Finally it loops through the clients and prints the data to the Console. The sample solution also includes a quick Windows Forms application (WinFormsApp) that demonstrates databinding use a DataGrid.

Data access is constant need for most business applications. Yet, the models and methods for doing data access are continually changing. The examples in this article should help you get started if you have chosen ADO.NET and Informix Dynamic Server as your tools.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Solution - all demo code</td>
<td>IfxAdo.zip (HTTP</td>
<td>FTP )</td>
</tr>
</tbody>
</table>
Related topics

- Download the Informix Client SDK.
- Download the free .NET Logging Framework from the ObjectGuy. Plenty of power without the complexity of other frameworks.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)