AllDb™ Software Development

When you're implementing with ADO.NET, you're dealing with data stores or, to use the better-known term, databases. Most developers are going to have to deal with the database (you as well, we assume, since you are reading this). If that thought frightens you, it shouldn't, because ADO.NET has made the database an easy thing to work with. The hard part now is no longer interfacing with the database, be it a 2-tier, 3-tier, or even n-tier architecture, but instead designing a good database.

ApexCalibur Software's one problem with ADO.NET is that it requires different code for each database that you want to implement, albeit mostly cut, paste and global replace. This is why we here at ApexCalibur Software decided to create AllDb™.

AllDb provides the functionality for implementing any ADO.NET supported database using a single set of source code[1]. The best part is that if you know how to code ADO.NET for Microsoft SQL Server 2000 then you have already completed 95% of your learning curve. AllDb uses, in almost all cases, the same syntax as Microsoft SQL Server 2000, even to the point of using its SQL dialect for embedded SQL[2].

If you don't know Microsoft's implementation of SQL, then you might need to read up on it a little bit. This document will not cover SQL syntax, as there are just simply to many differences between each of the ADO.NET supported databases. Remember, the focus of AllDb is a common single set of source code; you will still have to deal with creating the databases that you wish to support.

This document starts by covering a basic description of AllDb. It then moves on to describe the schema of the database used for all the examples in the document. It will point out some of the database implementation-specific differences that you will need to consider. Then, using this database, it will walk you through, using AllDb, the two methods provided by ADO.NET to access a database: connected and disconnected.

---

[1] In theory, AllDb™ should work with any .NET supported database because it uses a least common denominator approach. AllDb has been tested with six of the most commonly used .NET supported databases: **Microsoft SQL 2000**, **Oracle**, **DB2**, **Sybase SQL Anywhere**, **MySQL** and **Microsoft Access**.

[2] You cannot use any Microsoft specific commands within embedded SQL because this infringes on the least common denominator approach. This is not to say that you can't use implementation specific SQL within stored procedures, though this will require you to write equivalent code for all databases you support.

What Is AllDb™?

AllDb™ is a version 1.1 .NET Framework class library that allows you to access any ADO.NET supported database using a single piece of source code.

If you have provided multiple database support within the .NET Framework then you will have already experienced the fact that it requires duplicate source code for each database implementation. True, in many cases the duplication process is trivial, but as any veteran programmer can attest to, having to support multiple copies of a single implementation is, at the very least, a headache in the making since you have to keep all the source code in sync.

We here at ApexCalibur Software saw this headache as being unnecessary so we created AllDb.

AllDb is a set of classes that encompasses all aspects of accessing data sources within the ADO.NET architecture. It's designed to provide *least common denominator*[3] support for either connected or disconnected database access.

The classes that make up AllDb class library are all found within a single assembly named **AllDatabase.dll**[4]. You will have to add a reference to it in your application. After referencing the assembly, you will then have IntelliSense access to the classes and member methods and variables of AllDb's single namespace **ApexCalibur.data.AllDbClient**.

This document will go into detail about how to implement AllDb below but first it needs to cover how to create databases that AllDb can support.

Building AllDb Supported Databases

The key to developing AllDb supported databases is simplicity. Or, in other words, data access methods should be designed as simple queries (i.e., single SELECT, UPDATE, INSERT and DELETE). This will allow them to be implemented on all databases, usually with very little change. That isn't to say you can't be a little more elaborate, but remember that anything special or specific you implement to one database, you need to figure out how to implement for all databases.

---

[3] Due to the fact that there are so many different implementations of a database, the only way to support all databases is to support the features that are common to all. There are a few cases where a single database is missing a feature that all others support. In these cases, AllDb throws an AllDbException. It will be up to you to catch these exceptions and handle them.

[4] During the installation process, several other assemblies will be installed. These are required internally for AllDb to access other databases. AllDb uses product specific data adapters whenever possible to optimize its database access and some of these are not automatically installed with the base implementation of the .NET Framework.

Since you are restricted to simple queries, you now have to put a bit more effort in your business logic layer. For those of you who were used to adding, for example, conditional logic, calculations or local variables in your stored procedures, it may take a little getting used to, as you now have to handle it in your business logic layer.

Schema of the example database

The process of creating the actual database is beyond the scope of this document. Instead, we'll cover the basic schema and then a few database specific hints and gotchas of the example databases. The databases supported by the demo are:

- Microsoft SQL 2000
- Oracle 9
- MySQL 4.0[5]
- DB2 v8
- Sybase (Adaptive Server Anywhere 9)
- Access (OleDB).

The first thing unusual about these schemas is that you'll notice we use the .NET Data Type (AllDbType) to specify the database data type. The reason is that many databases represent these data types differently. The second is the Identity/Auto number — not all databases support this feature or, if they do, it might be in a roundabout way. We happen to know that the databases the demo supports also support Identity/Auto number in some fashion, so we were able to implement this feature. (You might want to reconsider this when you implement your own database.)

Author Database

A table of authors:

Name	AllDb Type	Length	Description	Identity / Auto Number	Primary Key	Allow Nulls
AuthorID	int	-	Auto generated ID number for author	Yes	Yes	No
FirstName	string	50	First name of the author	No	No	No
LastName	string	50	Last name of the author	No	No	No
MiddleName	string	50	Middle name of the author	No	No	Yes
Email	string	150	Email of the author	No	No	Yes

---

[5] If MySQL 5.0 is stable by the next release, then AllDb will support MySQL stored procedures.

Publisher

A table of publishers:

Name	AllDb Type	Length	Description	Identity / Auto Number	Primary Key	Allow Nulls
PublisherID	int	-	Auto generated ID number for publisher	Yes	Yes	No
Name	string	50	Name of the publisher	No	No	No
URL	string	100	URL of the publisher	No	No	Yes
Email	string	150	Email of the publisher	No	No	Yes

Books

A table of books with foreign keys to both the Author and Publisher databases:

Name	AllDb Type	Length	Description	Identity / Auto Number	Primary Key	Allow Nulls
ISBN	string	10	ISBN for the book	No	Yes	No
AuthorID	int	-	Foreign key to author table	No	No	No
PublisherID	int	-	Foreign key to publisher table	No	No	No
Title	string	100	Title of the book	No	No	No
Pages	int	-	Number of pages in the book	No	No	No
Price	Currency	-	Price of the book	No	No	No
DatePublished	DateTime	-	Date the book was published	No	No	No

Schema hints and gotchas

Oracle — The first gotcha that needs to be covered is that, to keep a consistent syntax within the AllDb source code, you need to create public synonyms for every item you create. That way you don't have to prefix everything with a User/Schema.

Oracle — Adding Identities or Auto-Numbering to a table is fairly straightforward for most of the supported databases. The exception is Oracle. Adding auto numbering requires the use of a SEQUENCE and a TRIGGER for each auto-numbering field. Fortunately, the code is simple and repetitive. Here is an example of the code for Auto-Numbering the Authors table:

```
CREATE SEQUENCE "ALLDBUSER"."SEQ_AUTHORS" INCREMENT BY 1 START WITH
    1 MAXVALUE 1.0E28 MINVALUE 1 NOCYCLE
    CACHE 100 NOORDER;
/
```

```
CREATE OR REPLACE TRIGGER "ALLDBUSER"."TRIG_AUTHORS"
BEFORE INSERT ON "ALLDBUSER"."AUTHORS"
FOR EACH ROW
BEGIN
  IF :new.AUTHORID IS NULL
    THEN SELECT SEQ_AUTHORS.nextval INTO :new.AUTHORID FROM DUAL;
  END IF;
END;
/
```

Access — Another thing to note is that you can't create a script for Access to add AutoNumber to a field. You need to change the field within the GUI designer.

Building Queries and Stored Procedures

With AllDb, you have four options when it comes to creating queries:

- Embed the code directly in the code (not really a good idea in all cases)
- Use Stored Procedures (not all database support this [most notably MySQL 4.1])
- Use a special feature of AllDb to place queries in a resource file
- Use a combination of Stored Procedures and resource file[6]

The demo program uses the fourth option since it covers the most databases.

Creating Stored Procedures

Most databases currently support stored procedures in some form. When creating stored procedures for AllDb, the key concepts are simplicity and trying to avoid database specific features. First, let's look at a couple of the stored procedures used in the demo.

The first stored procedure selects all columns for all rows from the table. The second stored procedure selects the row in the books table, which matches the passed ISBN parameter.

Oracle

```
CREATE OR REPLACE PROCEDURE "ALLDBUSER"."AUTHORS_SELECT"
  (IO_CURSOR OUT ALLDB_TYPES.ALLDB_CURSOR)
   AS
BEGIN
      OPEN IO_CURSOR FOR
      SELECT AUTHORID, FIRSTNAME, LASTNAME, MIDDLENAME, EMAIL
      FROM Authors;
END;
/

DROP PUBLIC SYNONYM AUTHORS_SELECT;
CREATE PUBLIC SYNONYM AUTHORS_SELECT FOR "ALLDBUSER"."AUTHORS_SELECT";

CREATE OR REPLACE PROCEDURE "ALLDBUSER"."Books_GetForISBN"
  (inISBN IN VARCHAR2, IO_CURSOR OUT ALLDB_TYPES.ALLDB_CURSOR)
AS
```

---

[6] You could include embedding into code here but using resources is a better and more versatile solution since you can use different resource files for different database. With embedded, you would need case or if statements.

```
BEGIN
      OPEN IO_CURSOR FOR
      Select *
      From BOOKS
      WHERE ISBN = inISBN;
END;
/

DROP PUBLIC SYNONYM Books_GetForISBN;
CREATE PUBLIC SYNONYM Books_GetForISBN FOR "ALLDBUSER"."Books_GetForISBN";
```

DB2

```
CREATE PROCEDURE Authors_Select ()
RESULT SETS 1
LANGUAGE SQL
BEGIN
      DECLARE c1 CURSOR FOR
      SELECT AUTHORID, FIRSTNAME, LASTNAME, MIDDLENAME, EMAIL
      FROM Adminstrator.Authors;
      OPEN c1;
END;

CREATE PROCEDURE Books_GetForISBN (IN @inISBN VARCHAR(20))
RESULT SETS 1
LANGUAGE SQL
BEGIN
      DECLARE c1 CURSOR FOR
      Select ISBN, AUTHORID, PUBLISHERID, TITLE, PAGES, PRICE, DATEPUBLISHED
      From BOOKS
      WHERE ISBN = @inISBN;
      OPEN c1;
END;
```

MS SQL 2000

```
CREATE PROCEDURE dbo.Authors_Select
AS
BEGIN
      SELECT AUTHORID, FIRSTNAME, LASTNAME, MIDDLENAME, EMAIL
        FROM Authors;
END
GO

CREATE PROCEDURE dbo.Books_GetForISBN
(
    @inISBN VARCHAR(20) = NULL
)
AS
BEGIN
      Select *
      From BOOKS
      WHERE ISBN = @inISBN;
END
GO
```

Stored Procedure hints and gotchas

The first thing you should notice is, at the core, the code for each of these databases is the same. Where they differ is how they handle the cursor.

MS SQL 2000 — If you have worked with MS SQL 2000, you know you don't have to worry about the cursor at all as its functionality is completely transparent.

DB2 — In the case of DB2, you only have to code for the cursor within the stored procedure itself. You don't have to do anything special within the AllDb code.

```
RESULT SETS 1
...
BEGIN
```

```
      DECLARE c1 CURSOR FOR
...
      OPEN c1;
End;
```

Oracle — Oracle's the 'problem child' of the three databases above. Not only do you have to code for it within the stored procedure, you also have to break standard SqlClient syntax within the AllDb code (by the way, it is the only noticeable deviation made by AllDb) by actually needing to declare a cursor parameter. We'll look at this in more detail when we cover AllDb code.

First, you have to create a package to create the cursor:

```
CREATE OR REPLACE PACKAGE "ALLDBUSER"."ALLDB_TYPES"
AS
 -- THIS FILE INCLUDES ALL REFERENCES NEEDED FOR ALLDB
      TYPE ALLDB_CURSOR IS REF CURSOR;
END;
/
```

Then you need to add the parameter declaration to the stored procedure and then open up the cursor:

```
CREATE OR REPLACE PROCEDURE "ALLDBUSER"."AUTHORS_SELECT"
  (IO_CURSOR OUT ALLDB_TYPES.ALLDB_CURSOR)
   AS
BEGIN
      OPEN IO_CURSOR FOR
...
END;
/
```

Finally, you have to create a public synonym:

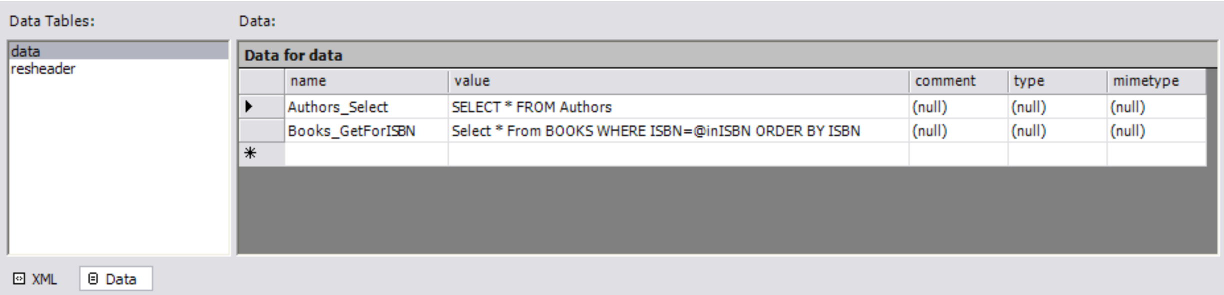
```
DROP PUBLIC SYNONYM AUTHORS_SELECT;
CREATE PUBLIC SYNONYM AUTHORS_SELECT FOR "ALLDBUSER"."AUTHORS_SELECT";
```

MS SQL 2000/DB2 — The only unusual thing (if you are used to coding stored procedures in MS SQL 2000 or DB2) is that you can't just use parameters of column names prefixed with the '@' symbol. Instead, you need to add some form of prefix (it really doesn't matter what. We use 'in'). The reason is that the '@' symbol gets stripped off for some databases (notice their absence in the Oracle version). Having identical column names and parameters, which for some databases is acceptable, makes things very hard to read and code.

Creating Resource File Queries

The procedure for creating resource file queries is really quite simple. Add a resource to your project and then, within the name column, place the name of the query (stored procedure) and, in that value column, place the actual query (stored procedure). Make sure you use the same query (stored procedure) names within the resource as you did in your code.

Once you have all your queries in the resource, the AllDb code will automatically use the value in the resource file if the database does not support stored procedures. (You can also force the use of the resource file by a property in AllDbControl class.)



	name	value	comment	type	mimetype
▶	Authors_Select	SELECT * FROM Authors	(null)	(null)	(null)
	Books_GetForISBN	Select * From BOOKS WHERE ISBN=@inISBN ORDER BY ISBN	(null)	(null)	(null)
*					

Figure 1. StoredProcedures.resx

Another feature of the resource file queries is that you don't need to declare the parameters unless you plan on using DeriveParameters. In this case, you simply prefix the query with the parameters enclosed in brackets:

```
(@inISBN String(10)) Select * From BOOKS WHERE ISBN = @inISBN
```

Resource File Queries Hints and Gotchas

When coding the queries in the resource file, use MS SQL 2000 syntax but avoid any of its special features or functions.

MySQL — We recommend avoiding, if at all possible, using the OUT direction of parameters unless you know you will not be supporting MySQL 4.1. This is because version 4.1 of MySQL and prior do not support this feature. (As of this writing, we don't know if version 5.0 supports the OUT direction but we would assume so as it is part of most standard stored procedure declarations.) This is probably one of the biggest gotchas of AllDb's lowest common denominator implementation. Hopefully, with MySQL 5.0 (and the next major release) this will no longer be an issue.

AllDb (sort of) Managed Provider

Managed providers provide ADO.NET with the capability to connect to and access from data sources. Their main purpose, as far as most developers are concerned, is to provide support for the DataAdapter class. This class is essentially for mapping between the data store and the DataSet.

Though AllDb appears in many ways to be a Managed Provider, in reality it is just a thin layer that uses actual Managed Providers. In fact, the first thing you do before you call any AllDb classes is tell which Managed Provider to use. You do this by setting the static parameter DBType in the AllDbControl[7] class. The DBType parameter is of type DatabaseType. AllDb version 1.0 supports the following types:

- Sql2000 — Microsoft SQL 2000
- Oracle — Oracle 9.0 (Probably works with 8i but it is untested)
- DB2 — DB2 version 8
- MySql — MySQL version 4.1
- Asa — Sybase (Adaptive Server Anywhere 9)
- OleDb — Tested with Access 2000
- Odbc — Tested with Access 2000

You can assign the DBType almost anywhere so long as it is before any calls to any other AllDb methods.

For web applications, we find the best place is in global.asax:

```
protected void Application_Start(Object sender, EventArgs e)
{
    AllDbControl.DBType = DatabaseType.MySql;
    AllDbControl.StoredProcResourceName = "CMSNET.StoredProcedures";
}
```

For simplicity, we hard coded the database type but we really don't recommend hard coding. It would probably better to declare an <appSettings> value in the web.config file and then grab the value out of it.

For a Windows application, you might consider placing the code in the main form's constructor or Load method. Again, declare the database to use in an <appSettings> value in the application.config file and then grab the value out of it.

In the code above, you will also notice how AllDb assigns the name of the Query resource file so that it can use it if the database does not support stored procedures or you purposely tell the code to use the resource file instead of the stored procedure:

```
AllDbControl.SupportsStoredProcedures = false;
```

---

[7] This is the sole class that doesn't have a Managed Provider equivalent. It contains several static members that are unique to AllDb.

Connected ADO.NET / AllDb

As stated previously, you have two distinct ways of accessing a database using ADO.NET. We cover the one that's easier to visualize and code first: connected access.

With connected access, you are continually connected to the database during the entire time you work with it. Like file access, you open the database, work with it for a while, and then you close it. Also like file I/O, you have the option of buffering data written to the database. This buffered access to the database is better known as *transactional database access.* We discuss this access method after we cover non-transactional database access.

Using Simple Connected ADO.NET

You'll start with the easiest way of working with the database, where the commands you execute happen immediately to the database.

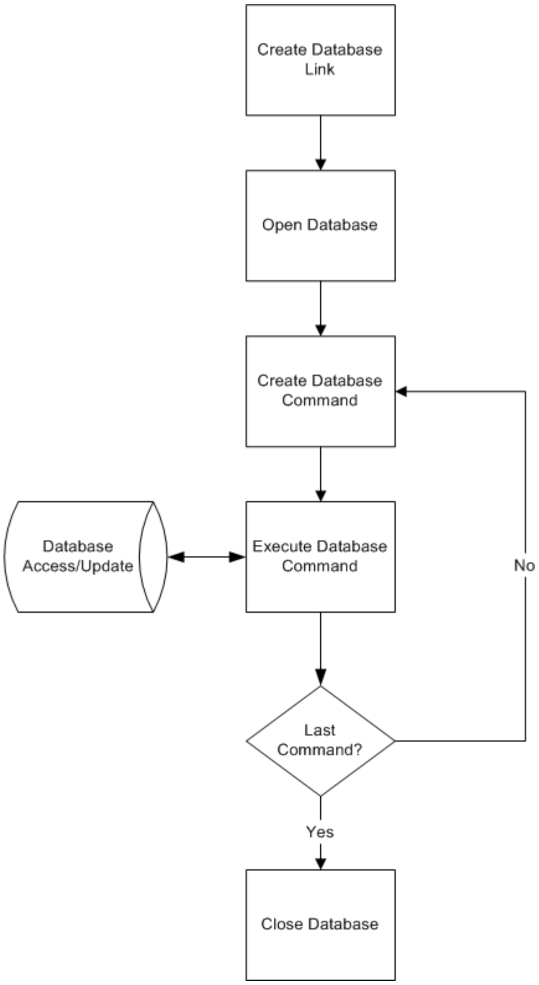


Figure 2. The basic flow of non-transactional database access.

1. Create a link to the database with an AllDbConnection.
2. Open the database with the Open() method.
3. Create a database command with AllDbCommand.
4. Execute the command by using one of the three methods within AllDbCommand (see Table 1). The database is immediately updated.
5. Repeat steps 3 and 4 until completed.
6. Close the database with the Close() method.

Table 1. The Main AllDbCommand SQL Statement Execution Methods

Method	Description
ExecuteNonQuery	Executes a statement that updates the database.
ExecuteReader	Executes a query to the database that could potentially return multiple rows from a database. This method returns an AllDbDataReader object that provides forward-only read access to the retrieved data or result set.
ExecuteScalar	Executes a statement that returns a single value.

Connecting, Opening, and Closing a Database

With connected non-transactional access to a database, you will always be connecting, opening, and closing your database. To handle this, you need to work with the Connection class: AllDbConnection.

Listing 1 shows how to connect, open, and close a database in a non-transactional method.

Listing 1. Connecting, Opening, and Closing a Database

```
using System;
using System.Data;

using ApexCalibur.Data.AllDbClient;

namespace Examples
{
    class Listing1
    {
        [STAThread]
        static void Main(string[] args)
        {
            string ConnectionString = @"SERVER=(local); DATABASE=AllDb; INTEGRATED SECURITY=True;";
            AllDbControl.DBType = DatabaseType.Sql2000;

//          string ConnectionString = @"SERVER=amidala; UID=system; PWD=password;";
//          AllDbControl.DBType = DatabaseType.Oracle;

//          string ConnectionString = @"SERVER=localhost; DATABASE=AllDb; UID=root; PWD=;";
//          AllDbControl.DBType = DatabaseType.MySql;

//          string ConnectionString = @"SERVER=localhost; DATABASE=AllDb; UID=administrator;” +
//                 “PWD=password;";
//          AllDbControl.DBType = DatabaseType.DB2;

//          string ConnectionString = @"Data Source=AllDb;UID=DBA;PWD=SQL";
//          AllDbControl.DBType = DatabaseType.Asa;

//          string ConnectionString = @"Provider=Microsoft.Jet.OLEDB.4.0; " +
//                  @"Data Source=D:\AllDb_Examples_CS\AllDb.mdb;Persist Security Info=False;";
//            AllDbControl.DBType = DatabaseType.OleDb;

            AllDbControl.StoredProcResourceName = "Examples.StoredProcedures";

            AllDbConnection connection = new AllDbConnection(ConnectionString);

            try
            {
                connection.Open();
                Console.WriteLine("We got a connection!");
            }
            catch (AllDbException e)
            {
                Console.WriteLine("No connection the following error occurred: {0}", e.Message);
            }
            finally
            {
                connection.Close();
                Console.WriteLine("The connection to the database has been closed");
            }
        }
    }
}
```

The first thing you do (as with any other .NET application) is import the namespaces needed to access AllDb's ADO.NET basic functionality:

```
using System;
using System.Data;
using ApexCalibur.Data.AllDbClient;
```

Next, as discussed above, you specify which database type you want to connect to and its connection string. In the example above we took the easy way out and hard coded all the combinations currently on a given computer. You will probably not hard code. Instead, you might want to place the appropriate values in some accessible spot like the application or web config files.

Note: From here on, we will only show the code for the setting of a single database type. (Got to save them trees!)

We added the code to attach the resource file to the AllDbControl. Though, in this example, it is never used.

```
AllDbControl.StoredProcResourceName = "Examples.StoredProcedures";
```

There is nothing special about creating an AllDbConnection class. It is just a constructor with a connection string parameter:

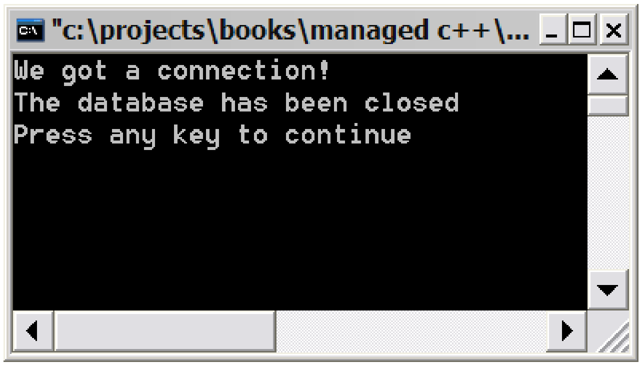
```
AllDbConnection connection = new AllDbConnection(ConnectionString);
```

Opening and closing the database is done virtually the same way as with a file, except the Open() method doesn't need any parameters:

```
connection.Open();
connection.Close();
```

You need to pay attention to the try statement. AllDb commands can abort in many places, so it is always a good thing to enclose your AllDb logic within a try clause and capture any exceptions by catching AllDbException.

It is also possible for AllDb to abort with the database still open. (Probably not in this example, but we felt having the correct code right from the beginning would make things clearer.) Therefore, it is a good idea to place your Close() method within a finally clause so that it will always be executed.



```
We got a connection!
The database has been closed
Press any key to continue
```

Figure 3. The results of the preceding example program. Impressive, no?

Querying a Database

All queries made to a connected database are done using the AllDbCommand class. As noted previously, the AllDbCommand class provides three methods to send SQL commands to the database, with each depending on the type of command. To query the database, you need to use the ExecuteReader() method.

Before you run the ExecuteReader() method, you need to configure AllDbCommand by placing the SQL command into it. There are two common ways of doing this. You can either place the SQL command, in text form, into the CommandText property, or you can place the name of the stored procedure containing the SQL command into the same property. The default method is the command in text form. If you plan to use a stored procedure, you need to change the CommandType property to StoredProcedure.

Listing 2 shows both methods. The first command uses a text-formatted command and retrieves the contents of the Authors table for authors with a specified LastName, in this case hard-coded to “Fraser”. The second command, using a stored procedure, retrieves all Books table records from a join with the Authors table where the Authors LastName equals the value passed to the stored procedure, in this case also “Fraser”.

Both calls to the ExecuteReader() method, after being configured, return an instance of AllDbDataReader, which is then iterated through to display the retrieved content.

Listing 2. The “Fraser” Books

```
using System;
using System.Data;

using ApexCalibur.Data.AllDbClient;

namespace Examples
{
    class Listing2
    {
        [STAThread]
        static void Main(string[] args)
        {
            string Name = "Fraser";

            string ConnectionString = @"SERVER=(local); DATABASE=AllDb; INTEGRATED SECURITY=True;";
            AllDbControl.DBType = DatabaseType.Sql2000;
            AllDbControl.StoredProcResourceName = "Examples.StoredProcedures";

            AllDbConnection connection = new AllDbConnection(ConnectionString);

            try
            {
                AllDbCommand cmd = new AllDbCommand();
                cmd.Connection = connection;

                cmd.CommandType = CommandType.Text;
                cmd.CommandText = string.Format("SELECT * FROM Authors WHERE LastName = '{0}'", Name);

                connection.Open();
                AllDbDataReader reader = cmd.ExecuteReader();
                while(reader.Read())
                {
                    Console.WriteLine("{0} {1} {2} -- {3}",
                        reader["FirstName"], reader["MiddleName"], reader["LastName"],
                        reader["Email"]);
                }
                reader.Close();

                //   SELECT   BOOKS.*
                //     FROM    BOOKS INNER JOIN
                //                    AUTHORS ON BOOKS.AUTHORID = AUTHORS.AUTHORID
                //    WHERE (AUTHORS.LASTNAME = @inLastName)

                cmd.CommandText = "Books_WhereLastName";       // This line must be before the next or
                cmd.CommandType = CommandType.StoredProcedure; // an exception will occur for databases
                                                               // that don't support stored procedures

                cmd.Parameters.Add(new AllDbParameter("@inLastName", AllDbType.String));
                cmd.Parameters["@inLastName"].Value = Name;

                cmd.Parameters.AddCursor();

                reader = cmd.ExecuteReader();

                Console.WriteLine("------------------------------------------------");

                while(reader.Read())
                {
                     Console.WriteLine(reader.GetString(0));
                     Console.WriteLine(reader.GetString(3));
                     Console.WriteLine(reader.GetInt32(4));
                     Console.WriteLine(reader.GetDecimal(5));
                     Console.WriteLine(reader.GetDateTime(6));
                     Console.WriteLine();
                 }
                reader.Close();
            }
            catch (AllDbException e)
            {
                Console.WriteLine("No connection the following error occurred: {0}", e.Message);
            }
            finally
            {
                connection.Close();
                Console.WriteLine("The connection to the database has been closed\n");
            }
        }
    }
}
```

The actual code to query a database with a CommandType of Text is pretty easy (if you know SQL, that is). First, you set the AllDbCommand class's CommandType property to Text:

```
cmd.CommandType = CommandType.Text;
```

Next, you place the SQL command you want to execute in the CommandText property. What makes this process easy is that you can use standard string formatting to build the command as you see here:

```
cmd.CommandText = string.Format("SELECT * FROM Authors WHERE LastName = '{0}'", Name);
```

Finally, you run the AllDbCommand class's ExecuteReader() method. This method returns an AllDbDataReader class from which you process the result set produced from the query:

```
AllDbDataReader reader = cmd.ExecuteReader();
```

Of course, you also need to open the connection before you execute the reader and close the connection after you finish processing the data.

The code to query a database with a CommandType of StoredProcedure is a little more difficult and has **two major gotchas**.

First, you place the name of the stored procedure you want to execute in the CommandText property:

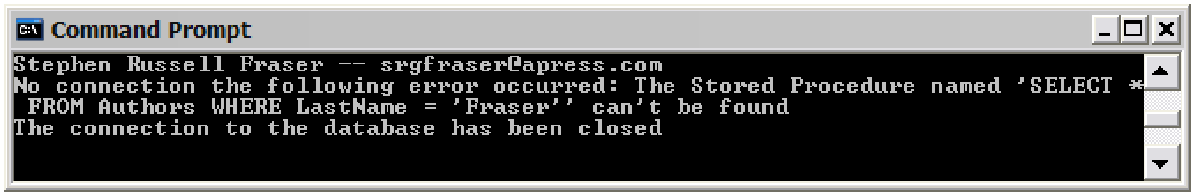
```
cmd.CommandText = "Books_WhereLastName";
```

Next, you set the AllDbCommand class's CommandType property to StoredProcedure:

```
cmd.CommandType = CommandType.StoredProcedure;
```

Caution: The order you do this in is important. AllDb internally needs the name of the stored procedure before the command type is set so that it can repopulate the command text with the value in the resource file if the currently implemented database does not support stored procedures.

If you forget to put these two commands in the right order, you will get a message similar to Figure 4.



```
Stephen Russell Fraser -- srgfraser@apress.com
No connection the following error occurred: The Stored Procedure named 'SELECT *
 FROM Authors WHERE LastName = 'Fraser'' can't be found
The connection to the database has been closed
```

Figure 4. Oops, wrong order.

If your stored procedure has parameters, you need to build a collection of AllDbParameters to place all the parameters that you want to pass to the stored procedure. The AllDbCommand class provides a property called Parameters to place your collection of AllDbParameters.

The first step is to use the Add() method off of the Parameters property collection to add all the AllDbParameters making up all the parameters that will be passed to the stored procedure. The constructor for the AllDbParameters class takes two or three parameters depending on the data type of the parameter that will be passed to the stored procedure. If the data type has a predefined length like int or a variable length like string, then only two parameters are needed.

```
cmd.Parameters.Add(new AllDbParameter("@inLastName", AllDbType.String));
```

Note: AllDb uses .NET Framework data types in the AllDbType enum and not database data types like the other managed providers.

When all the parameters are specified, you need to assign values to them so that the stored procedure can use them. You do this by assigning a value to the Value property of the appropriate index of the Parameters property collection of the AllDbCommand class. Clear as mud? This example should help:

```
cmd.Parameters["@inLastName"].Value = Name;
```

The second gotcha, as was mentioned above, is that to support Oracle databases you need to declare a cursor. To simplify things, AllDb added a new method to the AllDbParameterCollection class: AddCursor().

```
cmd.Parameters.AddCursor();
```

This simplification if the AllDb code came at the cost of making the stored procedure code a little more complex. The name of the cursor has been preset to an output direction parameter named IO_CURSOR. This means you must use the following SQL code in your stored procedures:

```
CREATE OR REPLACE PROCEDURE ...
([other parameters], IO_CURSOR OUT ALLDB_TYPES.ALLDB_CURSOR)
AS
BEGIN
    OPEN IO_CURSOR FOR
    ...
```

You need to also add the equivalent of the following package:

```
CREATE OR REPLACE PACKAGE "SCHEMA_NAME"."ALLDB_TYPES"
AS
    -- THIS FILE INCLUDES ALL REFERENCES NEEDED FOR ALLDB
    TYPE ALLDB_CURSOR IS REF CURSOR;
END;
```

Now, once you have done all that, you call the AllDbCommand class's ExecuteReader() method just like you did for a CommandType of Text:

```
reader = cmd.ExecuteReader();
```

The processing of the result set within the AllDbDataReader object is handled in a forward-only manner. The basic process is to advance to the next record of the result set using the Read() method. If the return value is false, you have reached the end of the result set and you should the call the Close() method to close the AllDbDataReader. If the value is true, then you continue and process the next result set record.

```
while(reader.Read())
{
        Console.WriteLine("{0} {1} {2} -- {3}",
        reader["FirstName"], reader["MiddleName"], reader["LastName"], reader["Email"]);
}
reader.Close();
```

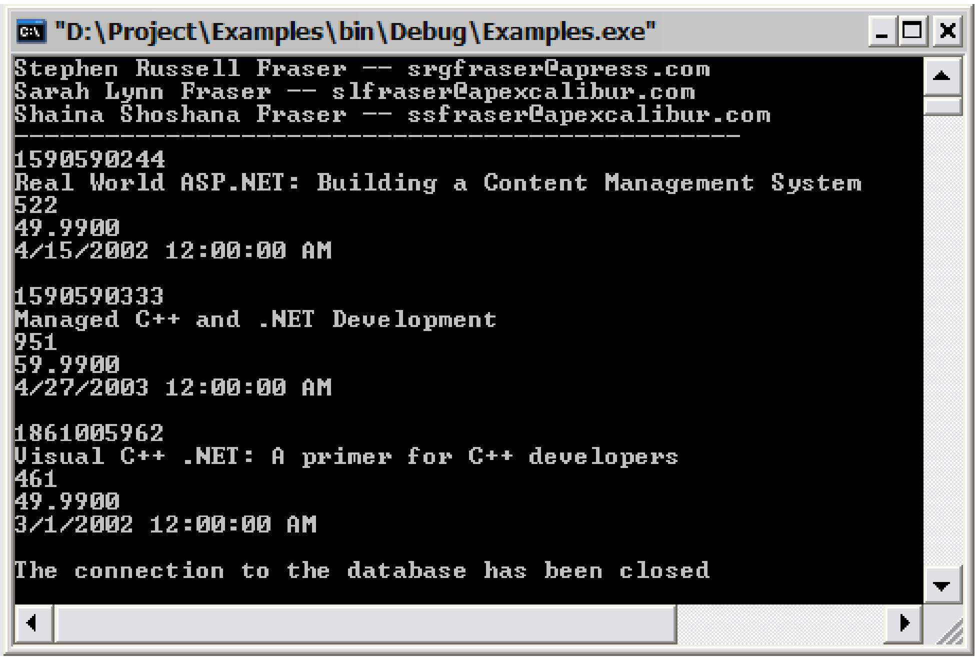
There are two different methods of processing the record set. You can, as we did, use the indexed property to get the value based on the column header. You can also process the columns using an assortment of type-specific Getxxx() methods where xxx is the data type for which you want to receive the data:

```
while(reader.Read())
```

```
{
    Console.WriteLine(reader.GetString(0));
    Console.WriteLine(reader.GetString(3));
    Console.WriteLine(reader.GetInt32(4));
    Console.WriteLine(reader.GetDecimal(5));
    Console.WriteLine(reader.GetDateTime(6));
    Console.WriteLine();
}
reader.Close();
```

Note the parameter passed in the position of the column starting at zero. You also have to know the order of the columns so you can assign the correct ordinals.

We find using column names easier, but the style you choose to use is up to you.



```
Stephen Russell Fraser -- srgfraser@apress.com
Sarah Lynn Fraser -- slfraser@apexcalibur.com
Shaina Shoshana Fraser -- ssfraser@apexcalibur.com
------------------------------------------------
1590590244
Real World ASP.NET: Building a Content Management System
522
49.9900
4/15/2002 12:00:00 AM

1590590333
Managed C++ and .NET Development
951
59.9900
4/27/2003 12:00:00 AM

1861005962
Visual C++ .NET: A primer for C++ developers
461
49.9900
3/1/2002 12:00:00 AM

The connection to the database has been closed
```

Figure 5. Retrieving Fraser's books.

Insert, Update, and Delete Commands

The code to modify the database (i.e., insert, update, and delete rows of the database) isn't much different from the code to query the database. Obviously, the SQL is different. The only other difference is that you call the AllDbCommands class's ExecuteNonQuery() method instead of the ExecuteReader() method.

You can still use both CommandTypes and you still need to set up the AllDbParameters the same way for stored procedures.

In Listing 3 you insert a new record into the database, you change the Last Name on the record, and then you delete the record. (A lot of work for nothing, don't you think?)

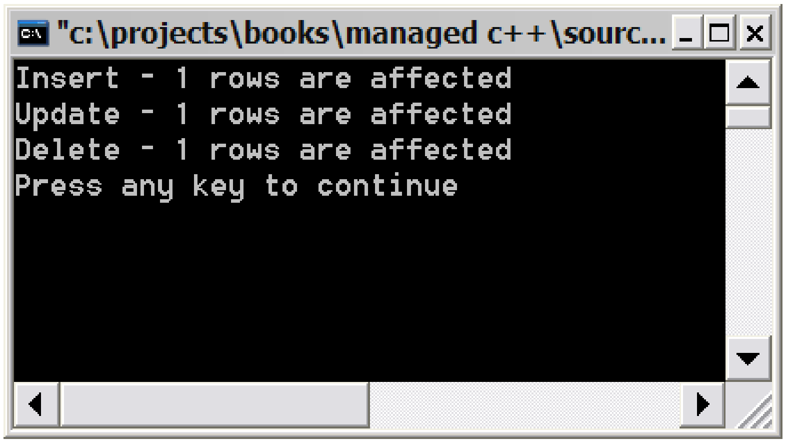
Listing 3. Modifying the Database

```
using System;
using System.Data;

using ApexCalibur.Data.AllDbClient;

namespace Examples
```

```
{
    class Listing3
    {
        [STAThread]
        static void Main(string[] args)
        {
            string ConnectionString = @"SERVER=(local); DATABASE=AllDb; INTEGRATED SECURITY=True;";
            AllDbControl.DBType = DatabaseType.Sql2000;
            AllDbControl.StoredProcResourceName = "Examples.StoredProcedures";

            AllDbConnection connection = new AllDbConnection(ConnectionString);

            try
            {
                AllDbCommand cmd = new AllDbCommand();
                cmd.Connection = connection;
                connection.Open();

                //    INSERT INTO AUTHORS
                //           (FirstName, LastName, MiddleName, Email)
                //    VALUES (inFirstName, inLastName, inMiddleName, inEmail);

                cmd.CommandText = "InsertAuthor";
                cmd.CommandType = CommandType.StoredProcedure;

                cmd.Parameters.Add(new AllDbParameter("@inFirstName", AllDbType.String, 50));
                cmd.Parameters.Add(new AllDbParameter("@inLastName", AllDbType.String, 50));
                cmd.Parameters.Add(new AllDbParameter("@inMiddleName", AllDbType.String, 50));
                cmd.Parameters.Add(new AllDbParameter("@inEmail", AllDbType.String, 150));

                cmd.Parameters["@inFirstName"].Value = "John";
                cmd.Parameters["@inLastName"].Value  = "Dope";
                cmd.Parameters["@inMiddleName"].Value  = DBNull.Value;  // Some databases don't
                cmd.Parameters["@inEmail"].Value  = DBNull.Value;       // support default values

                int affected = cmd.ExecuteNonQuery();
                Console.WriteLine("Insert - {0} rows are affected", affected);

                cmd.Parameters.Clear();    // Leaving old parameters confuses some databases

                cmd.CommandText = "UPDATE Authors SET LastName = 'Doe' WHERE LastName = 'Dope'";
                cmd.CommandType = CommandType.Text;

                affected = cmd.ExecuteNonQuery();
                Console.WriteLine("Update - {0} rows are affected", affected);

                cmd.CommandText = "DELETE FROM Authors WHERE LastName = 'Doe'";
                cmd.CommandType = CommandType.Text;

                affected = cmd.ExecuteNonQuery();
                Console.WriteLine("Delete - {0} rows are affected", affected);
            }
            catch (AllDbException e)
            {
                Console.WriteLine("The following error occurred: {0}",
                    e.Message);
            }
            finally
            {
                connection.Close();
                Console.WriteLine("\nThe connection to the database has been closed\n");
            }
        }
    }
}
```

As you can see, there is not much new going on here in the AllDb code, other than the call to `ExecuteNonQuery()`. This method returns the number of rows affected by the SQL command.

```
int affected = cmd.ExecuteNonQuery();
```



```
Insert - 1 rows are affected
Update - 1 rows are affected
Delete - 1 rows are affected
Press any key to continue
```

Figure 6. A lot of modifications to the database for no gain.

Returning a Single Value from a Query

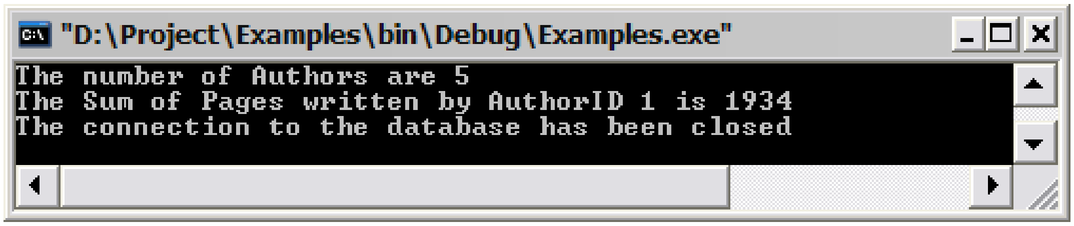
The final command executing method of the `AllDbCommand` class is `ExecuteScalar()`. This method is design to return an `object` pointer as the result of the query. The returned `object` points to a value like that produced by an aggregated SQL function such as `COUNT` or `SUM`. Again, like the database modifying command, there is not much changed between the source code needed to execute this type of method and that of a standard query.

Listing 4 shows how to count all the records in the author database and also how to sum a column. In this case, it is the number of pages written by the author specified by ID.

Listing 4. Counting and Summing

```
using System;
using System.Data;

using ApexCalibur.Data.AllDbClient;

namespace Examples
{
    class Listing4
    {
        [STAThread]
        static void Main(string[] args)
        {
            string ConnectionString = @"SERVER=(local); DATABASE=AllDb; INTEGRATED SECURITY=True;";
            AllDbControl.DBType = DatabaseType.Sql2000;

            AllDbControl.StoredProcResourceName = "Examples.StoredProcedures";

            AllDbConnection connection = new AllDbConnection(ConnectionString);

            try
            {
                AllDbCommand cmd = new AllDbCommand();
                cmd.Connection = connection;

                connection.Open();

                cmd.CommandType = CommandType.Text;
                cmd.CommandText = "SELECT COUNT(*) FROM Authors";

                object NumAuthors = cmd.ExecuteScalar();
                Console.WriteLine("The number of Authors are {0}", NumAuthors);

                //  SELECT SUM(Pages) FROM Books
                //   WHERE AuthorID = @inAuthorID
                cmd.CommandText = "PagesByAuthorID";
                cmd.CommandType = CommandType.StoredProcedure;

                cmd.Parameters.Add(new AllDbParameter("@inAuthorID", AllDbType.Int32));
                cmd.Parameters["@inAuthorID"].Value = 1;

                cmd.Parameters.AddCursor();

                object TotalPages = cmd.ExecuteScalar();
                Console.WriteLine("The Sum of Pages written by AuthorID {0} is {1}", 1, TotalPages);
            }
            catch (AllDbException e)
            {
                Console.WriteLine("The following error occurred: {0}",
                    e.Message);
            }
            finally
            {
                connection.Close();
                Console.WriteLine("The connection to the database has been closed\n");
            }
        }
    }
}
```

As you can see, other than the SQL code and the calling of the `ExecuteScalar()` method, there is not much new. The `ExecuteScalar()` method returns a pointer to an `object`, which you can typecast to the type of the return value. In both cases, you could have typecast the return `object` pointer to `Int32`, but the `WriteLine()` method can do it for you.



```
The number of Authors are 5
The Sum of Pages written by AuthorID 1 is 1934
The connection to the database has been closed
```

Figure 7. Counting rows and summing a column.

Using Connected ADO.NET with Transactions

Think about this scenario. You buy a computer on your debit card, but while the purchase is being processed, the connection to the debit card company is lost. The response from the debit card reader is a failure message. You try again and the debit card reader now responds that there's not enough money. You go home empty-handed, angry, and confused. Then, a month later, your bank statement says you bought a computer with your debit card.

It can't happen, right? Wrong. If you use the preceding immediate updating method, it's very possible, as each update to the database is stand-alone. One command can complete, for example, the withdrawal, while a second command may fail, for example, the sale.

This is where transactions come in handy. They make sure all database commands needed to complete a process are completed successfully before allowing the database to commit (or write) these commands. If one or more of the commands fail, the database can reject all of the commands and return to its original state before any of the commands were completed. This is known as *rolling back.*

Note: With MySQL, you need to create your table with a type that supports transactions (either InnoDB or BDB). This is done by adding the Table Create optional: Type=<database type>, to the end of the database create statement. If you forget to do this, you will get an AllDbException stating: "Warning: Some non-transactional changed tables couldn't be rolled back".

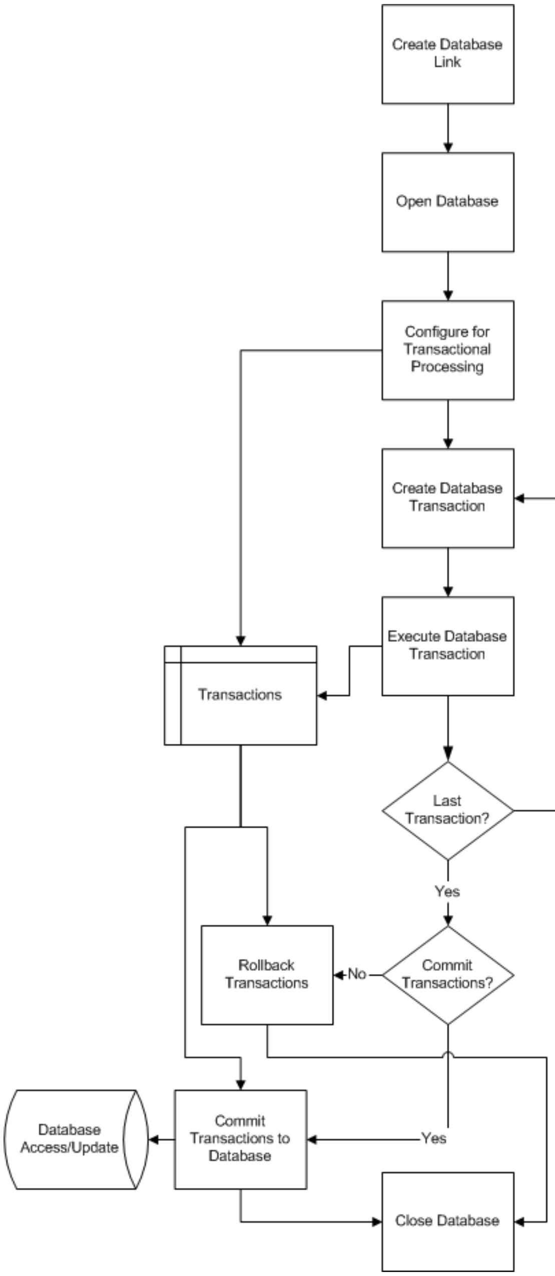


Figure 8. The basic flow of transactional database access.

Here is the basic flow of transactional database access:

1. Create a link to the database with an AllDbConnection.
2. Open the database with the Open() method.
3. Configure for transactions.
4. Create a database transaction with AllDbCommand.
5. Execute the transaction by using the ExecuteNonQuery() method of the AllDbCommand class. The temporary copy of the database is updated.
6. Repeat steps 4 and 5 until completed.
7. When all transactions complete, either commit the transactions to the database or roll them back.
8. Close the database with the Close() method.

Listing 5 shows how to convert the non-transactional example from Listing 3, which we covered above, into a transactional example.

Listing 5. Transactional Database Updates

```
using System;
using System.Data;

using ApexCalibur.Data.AllDbClient;

namespace Examples
{
    class Listing3
    {
        [STAThread]
        static void Main(string[] args)
        {
            string ConnectionString = @"SERVER=(local); DATABASE=AllDb; INTEGRATED SECURITY=True;";
            AllDbControl.DBType = DatabaseType.Sql2000;

            AllDbControl.StoredProcResourceName = "Examples.StoredProcedures";

            AllDbConnection connection = new AllDbConnection(ConnectionString);

            AllDbTransaction transaction = null;
            AllDbCommand cmd = new AllDbCommand();

            try
            {
                cmd.Connection = connection;

                connection.Open();   // Must be open before beginning transactions

                transaction = connection.BeginTransaction();
                cmd.Transaction = transaction;

                //    INSERT INTO AUTHORS
                //           (FirstName, LastName, MiddleName, Email)
                //    VALUES (inFirstName, inLastName, inMiddleName, inEmail);

                cmd.CommandText = "InsertAuthor";
                cmd.CommandType = CommandType.StoredProcedure;

                cmd.Parameters.Add(new AllDbParameter("@inFirstName", AllDbType.String, 50));
                cmd.Parameters.Add(new AllDbParameter("@inLastName", AllDbType.String, 50));
                cmd.Parameters.Add(new AllDbParameter("@inMiddleName", AllDbType.String, 50));
                cmd.Parameters.Add(new AllDbParameter("@inEmail", AllDbType.String, 150));

                cmd.Parameters["@inFirstName"].Value = "John";
                cmd.Parameters["@inLastName"].Value  = "Dope";
                cmd.Parameters["@inMiddleName"].Value  = DBNull.Value;  // Some DBs don't support
                cmd.Parameters["@inEmail"].Value  = DBNull.Value;       // default values

                int affected = cmd.ExecuteNonQuery();

                if (affected <= 0 && AllDbControl.DBType != DatabaseType.DB2) // DB2 doesn't return
                    throw new Exception("Insert Failed");                     // a count on Update

                Console.WriteLine("Insert - {0} rows are affected", affected);
                cmd.Parameters.Clear();    // Leaving old parameters confuses some databases

                cmd.CommandText = "UPDATE Authors SET LastName = 'Doe' WHERE LastName = 'Dope'";
                cmd.CommandType = CommandType.Text;

                affected = cmd.ExecuteNonQuery();

                if (affected <= 0)
                    throw new Exception("Update Failed");

                Console.WriteLine("Update - {0} rows are affected", affected);

                // This transaction will return 0 affected rows
                // because "Does" does not exist.
                // Thus, the following 'if condition' throws an exception which causes all
                // Transactions to be rolled back.

                cmd.CommandText = "DELETE FROM Authors WHERE LastName = 'Does'";
                cmd.CommandType = CommandType.Text;

                affected = cmd.ExecuteNonQuery();

                if (affected <= 0)
                    throw new Exception("Delete Failed");

               Console.WriteLine("Delete - {0} rows are affected", affected);

                transaction.Commit();
                Console.WriteLine("Transaction completed -- Committed changes");
            }
            catch (Exception e)
            {
                try
                {

                    if (transaction != null)
                        transaction.Rollback();

                    Console.WriteLine("The following error occurred: {0}\n", e.Message);
                    Console.WriteLine("Transaction Not completed -- Rolled back changes");
                }
                catch (Exception ex)
                {
                    if (transaction.Connection != null)
                    {
                        // You need to select a table type that supports
                        // transactions probably either InnoDB or BDB
                        Console.WriteLine("An exception of type " + ex.GetType() +
                            " was encountered while attempting to roll back the transaction.");
                        Console.WriteLine("Error: " + ex.Message);
                    }
                }
            }
            finally
            {
                connection.Close();
                Console.WriteLine("\nThe connection to the database has been closed\n");
            }
        }
    }
}
```

As you can see, there have not been many changes. We added a few statements to capture that a transaction failed. This was simply done by adding “if conditions” after each command execution checking to see if the number of rows processed was greater than 1.

> **Caution** For some reason DB2 “Insert Command” returns –1 instead of the number of rows processed. We are not aware of any work around at this time for our code. [If someone is, (I’m sure there is) please submit it to ApexCalibur.com and we’ll update as appropriate.]

To actually implement transactional code, though, you must first declare an AllDbTransaction class:

```
AllDbTransaction transaction;
```

Next, you need to create a transaction set using the AllDbConnection class's BeginTransaction() method. You must be careful where you place this statement as it must be placed after the connection has been opened or you will get some form of "Invalid Operation" exception depending on the database.

```
transaction = connection.BeginTransaction();
```

Now that you have a transaction set, you need to assign it to the AllDbCommand class's property Transaction. Again if you fail to do this, some databases, depending on how they implemented Transactions internally, will throw up some form of exception.

```
cmd.Transaction = transaction;
```

The last step of transactional database update is to commit all the transactions upon the database when all is well. Therefore, if everything completed successfully, execute the AllDbTransaction class's Commit() method.

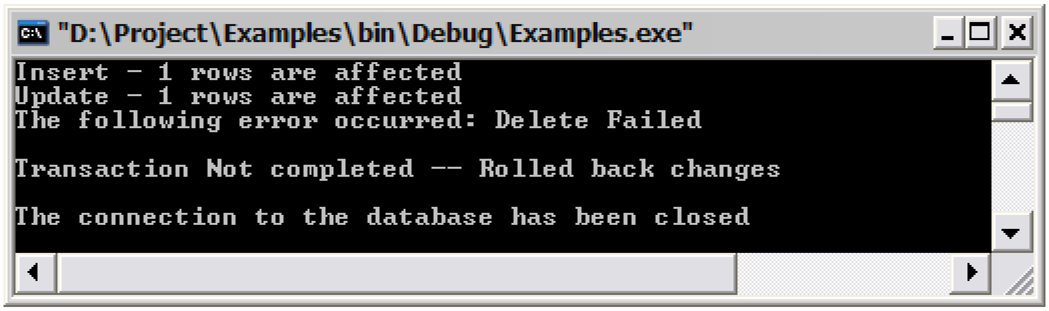
```
transaction->Commit();
```

If, on the other hand, an error occurs, you would then execute the AllDbTransaction class's Rollback() method:

```
transaction->Rollback();
```

We implemented the above code so that if something goes wrong, an exception is thrown. With this style of implementation, we can safely place the commit and rollback commands each in one place. The commit at the end of all transactions and the rollback within the exception's catch clause.

Figure 9 shows a successfully failed Transaction. Or is that a Transaction that failed successfully?



```
Insert - 1 rows are affected
Update - 1 rows are affected
The following error occurred: Delete Failed

Transaction Not completed -- Rolled back changes

The connection to the database has been closed
```

Figure 9. Transactional database update rollback.

Disconnected ADO.NET / ALLDb

Let's switch gears now and look at disconnected ADO.NET. Disconnected data access is a key feature of ADO.NET. Basically, it means that most of the time when you're accessing a database, you aren't getting the data from the database at all. Instead, you're accessing a synchronized, in-memory copy of the data that was moved earlier to your client computer. Don't worry about all the technical issues surrounding this, just be glad that it works because it provides three major benefits:

- Less congestion on the database server because users are spending less time connected to it.
- Faster access to the data because the data is already on the client.
- Works across disconnection networks such as the Internet.

It also offers one benefit (associated with a disconnect access) that is less obvious: Data doesn't have to be stored in a database-like format. Realizing this, Microsoft decided to implement ADO.NET using a strong typed XML format. The benefit is that having data in XML format enables data to be transmitted using standard HTTP. This yields a further benefit: Firewall problems disappear. An HTTP response with the body of XML flows freely through a firewall, unlike the pre-ADO.NET technology's system-level COM marshalling requests. If the previous bonus is Greek (or geek) to you, don't fret. In fact, be glad you have no idea what we're talking about.

The Core Classes

If you spend a lot of time working with ADO.NET, you may have an opportunity to work with almost all of ADO.NET's classes. For the purpose of this document, however, we've trimmed these classes down to the following:

- AllDbDataAdaptor
- DataSet
- DataTable
- DataRow
- DataColumn
- DataRelation
- Constraint

All of these classes interact with each other in some way. Figure 10 shows the flow of the interaction. Essentially, the AllDbDataAdaptor connects the data store to the DataSet. The DataSet stores the data in a Tables property containing a DataTablesCollection made up of one or more DataTables. Each DataTable is made up of DataRows and DataColumns. All of the DataTables store their relationships in a Relations property containing a DataRelationCollection made up of DataRelations. Finally, each DataTable can be affected by Constraints. Simple, no?

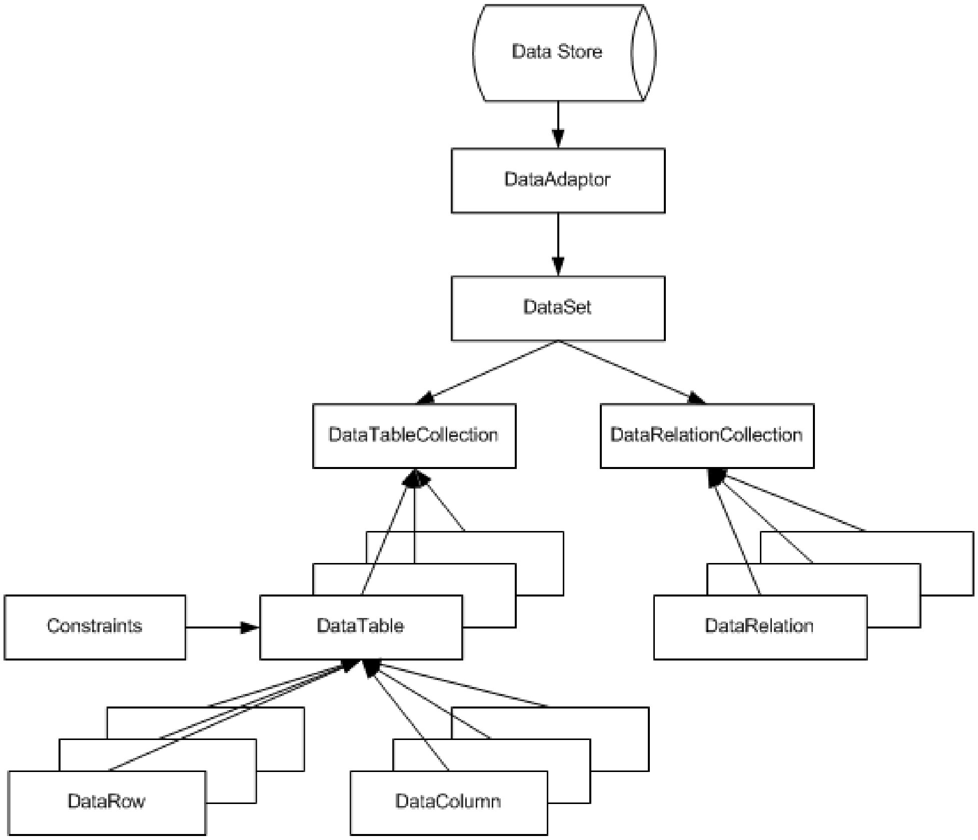


Figure 10. The disconnected ADO.NET class interaction.

AllDbDataAdaptor

The AllDbDataAdaptor is the bridge between a data source (database) and the DataSet. Its purpose is to extract data out of the data source and place it in the DataSet. Then it updates, if required, the data source with the changes made in the DataSet.

It should be relatively easy to get comfortable with the AllDbDataAdaptor, as it uses the (just like connected database access) connection class, AllDbConnection, to connect to the data source and a command class, AllDbCommand, to add, update, and select data out of the data source.

The basic idea behind using the AllDbDataAdaptor is to provide SQL commands to the following four properties to handle sending and receiving data between the DataSet and the data store:

* SelectCommand
* InsertCommand
* UpdateCommand
* DeleteCommand

If you plan to only read data from the database, then only the SelectCommand property needs to be provided.

With these properties provided, it is a simple matter to call the AllDbDataAdaptor class's Fill() method to select data from the data store to the DataSet and to call the Update() method to insert, update, and/or delete data from the DataSet to the data store.

DataSet Class

The DataSet is the major controlling class for disconnected ADO.NET. A DataSet is a memory cache used to store all data retrieved from a data source, in most cases a database or XML file. The data source is connected to the DataSet using an AllDbDataAdaptor.

A DataSet consists of one or more data tables in a DataTableCollection class, which in turn is made up of data rows and data columns. Relationships between the tables are maintained via a DataRelationsCollection class. The DataSet also stores the format information about the data.

A DataSet also provides transactional access to its data. To commit all changes made to the DataSet from the time it was created or since the last time it was committed, call the DataSet class's AcceptChanges() method. If you want to roll back changes since the DataSet was corrected or since it was last committed, call the RejectChanges() method. What actually happens is a cascading effect where the AcceptChanges() and RejectChanges() methods execute their table's versions of the method, which in turn calls the table's row's version. Thus, it is also possible to commit or rollback at the table and row levels.

DataTableCollection Class

A DataTableCollection is a standard collection class made up of one or more DataTables. Like any other collection class, it has functions such as Add, Remove, and Clear. Usually, you will not use any of this functionality. Instead, you will use it to get access to a DataTable stored in the collection.

The method of choice for doing this will probably be to access the DataTableCollection's indexed property, using the name of the table that you want to access as the index:

```
DataTable dt = dataSet.Tables["Authors"];
```

It is also possible to access the same table using the overloaded array property version as well:

```
DataTable dt = dataset.Tables[0];
```

With this method, you need to know which index is associated with which table. When you use the indexed property, it is a little more obvious.

DataTable Class

Put simply, a DataTable is one table of data stored in memory. A DataTable also contains constraints, which help ensure the integrity of the data it is storing.

It should be noted that a DataTable could be made up of zero or more DataRows, because it is possible to have an empty table. Even if the table is empty, the Columns property will still contain a collection of the headers that make up the table.

Many properties and methods are available in the DataTable, but in most cases you will simply use it to get access to the rows of the table. Three of the most common methods are enumerating through the Rows collection:

```
IEnumerator Enum = dt.Rows.GetEnumerator();
while(Enum.MoveNext())
{
    DataRow row = (DataRow)Enum.Current;
    // ... Do stuff to row
}
```

using the foreach to walk through:

```
foreach (DataRow row in dt.Rows)
{
    // ... Do stuff to each row
}
```

and selecting an array of DataRows using the Select() method:

```
DataRow row[] =  dt.Select(string.Format(S"AuthorID={0}", CurrentAuthorID));
```

Another method that you will probably come across is NewRow(), which creates a new DataRow, which will later be added to the DataTable Rows collection:

```
DataRow row = dt.NewRow();
// ... Build row
dt.Rows.Add(row);
```

DataRow Class

The DataRow is where the data is actually stored. You will frequently access the data from the DataRow as indexed property, using the name of the column that you want to access as the index.

```
Row["LastName"] = tbLastName.Text;
```

It is also possible to access the same column using the overloaded array property version:

```
row[0] = tbLastName.Text;
```

With this method, you need to know which index is associated with which column. When you use the indexed property, it is a little more obvious.

DataColumn Class

You use the DataColumn class to define the columns in a DataTable. Each DataColumn has a data type that determines the kind of data it can hold. A DataColumn also has properties similar to a database, such as AllowNull and Unique. If the DataColumn auto-increments, then the AutoIncrement property is set.

One gotcha that could get you is that some databases return their columns as ReadOnly. If you plan on updating the columns, you will get an exception. When this occurs, you need to go through all the columns and set them to being Writable. To do this, simply set all the column's ReadOnly properties to false. To simplify things, AllDb adds a static method in the AllDbControl class to do this for you:

```
DataTable dt = dSet.Tables["Authors"];

DataRow[] row = dt.Select(string.Format("AuthorID={0}",
                ((ListBoxItem)lbAuthors.SelectedItem).row["AuthorID"]));

AllDbControl.MakeTableWritable(dt); // Some databases retreive columns as Readonly
```

DataRelationCollection Class

A DataRelationCollection is a standard collection class made up of one or more DataRelations. Like any other collection class, it has functions such as Add, Remove, and Clear. Usually, as with the DataTableCollection class, you will probably not use any of this functionality. Instead, you will simply use it to get access to the DataRelations it stores.

DataRelation Class

A DataRelation is used to relate two DataTables together. It does this by matching DataColumns between two tables. You can almost think of it as the ADO.NET equivalent of the foreign-key relationship in a relational database (like you previously set).

One important thing you have to keep in mind is that the DataColumns must be the same data type. Remember that ADO.NET has strong data types, and when comparing different data types, one data type must be converted to the other. This conversion is not done automatically.

Constraint Classes

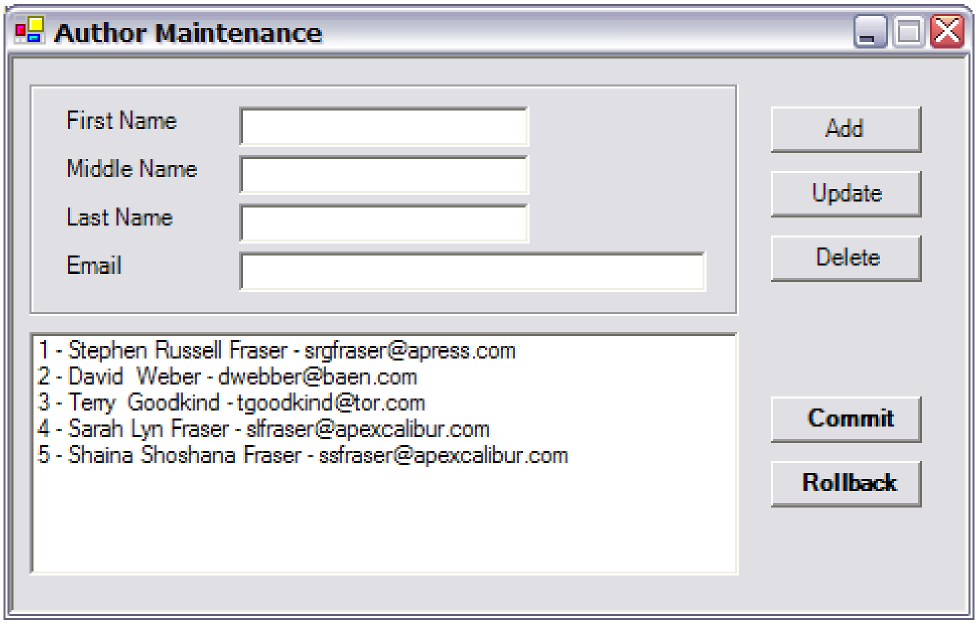
The Constraint classes make it possible to add a set of constraints on a particular column in your DataTable. Two types of constraints are currently supported by ADO.NET:

- ForeignKeyConstraint disallows a row to be entered unless there is a matching row in another (parent) table.
- UniqueConstraint makes sure that a column is unique within a DataTable.

Developing with Disconnected ADO.NET

In the final example, you're going to build a small Win Form application to maintain the Authors table that you've been working with throughout the document. The example uses disconnected data source access with full select, insert, update, and delete capabilities that can be either committed or rolled back.

A good portion of the code is related to Win Forms and is not included here. What you will see in the example is the code that wasn't auto-generated by Visual Studio .NET. Figure 11 shows the final result of the example, from which you can build your own Win Form.



1 - Stephen Russell Fraser - srgfraser@apress.com
2 - David Weber - dwebber@baen.com
3 - Terry Goodkind - tgoodkind@tor.com
4 - Sarah Lyn Fraser - slfraser@apexcalibur.com
5 - Shaina Shoshana Fraser - ssfraser@apexcalibur.com

Figure 12-13. The Author Maintenance tool.

Building the AllDbDataAdaptor

The first thing that you need to do is build the application's AllDbDataAdaptor. Then you'll use the AllDbDataAdaptor to place data in the DataSet. Eight major steps (three of which are optional) are involved in building an AllDbDataAdaptor and populating and maintaining a DataSet:

1. Create an AllDbConnection.
2. Create an AllDbDataAdaptor.
3. Implement a SelectCommand property.
4. Implement an InsertCommand property (optional).
5. Implement an UpdateCommand property (optional).
6. Implement a DeleteCommand property (optional).
7. Create a DataSet.

8. Populate (fill) the DataSet.

Creating an AllDbConnection is done the same way as you did with connected database access:

```
string ConnectionString = @"SERVER=(local); DATABASE=AllDb; INTEGRATED SECURITY=True;";
AllDbControl.DBType = DatabaseType.Sql2000;

// string ConnectionString = @"SERVER=amidala; UID=system; PWD=password;";
// AllDbControl.DBType = DatabaseType.Oracle;
//
// string ConnectionString = @"SERVER=localhost; DATABASE=AllDb; UID=root; PWD=;";
// AllDbControl.DBType = DatabaseType.MySql;
//
// string ConnectionString = @"SERVER=localhost; DATABASE=AllDb; UID=administrator; PWD=password;";
// AllDbControl.DBType = DatabaseType.DB2;
//
// string ConnectionString = @"Data Source=AllDb;UID=DBA;PWD=SQL";
// AllDbControl.DBType = DatabaseType.Asa;
//
// string ConnectionString = @"Provider=Microsoft.Jet.OLEDB.4.0;" +
//      @"Data Source=D:\Project\AllDbTester\AllDb.mdb;Persist Security Info=False;";
// AllDbControl.DBType = DatabaseType.OleDb;

AllDbControl.StoredProcResourceName = "Listing6.StoredProcedures";

AllDbConnection connect = new AllDbConnection(ConnectionString);
```

Creating the AllDbDataAdapter is a simple constructor call. You probably want to also add the primary key information. This ensures that incoming records that match existing records are updated instead of appended:

```
dAdapt = new AllDbDataAdapter();
dAdapt.MissingSchemaAction = MissingSchemaAction.AddWithKey;
```

The SelectCommand is the SQL command that will be used to populate the DataSet. It can be as complex or as simple as you like. The implementation of the SelectCommand requires a standard AllDbCommand like the one you created earlier with connected access. Notice two things. First, the constructor takes the stored procedure name or the command itself and the data source connection. Second, to support Oracle, you need to implement the AddCursor() method:

```
dAdapt.SelectCommand = new AllDbCommand("Authors_Select", connect);
dAdapt.SelectCommand.CommandType = CommandType.StoredProcedure;

dAdapt.SelectCommand.Parameters.AddCursor();
```

The InsertCommand is the SQL command that will be executed to insert added DataSet rows back into the data source. The implementation of this property is a little tricky, as it requires parameters to be passed to the command. The Add() method to the Parameters property is similar to what you have seen previously, except it has one additional parameter and the size parameter is mandatory, even if it is obvious, as in the case of Int32. The additional property is the name of the column that the data will be extracted from:

```
dAdapt.InsertCommand = new AllDbCommand("Authors_Insert", connect);
dAdapt.InsertCommand.CommandType = CommandType.StoredProcedure;

dAdapt.InsertCommand.Parameters.Add("@inFirstName", AllDbType.String, 50, "FirstName");
dAdapt.InsertCommand.Parameters.Add("@inMiddleName", AllDbType.String, 50, "MiddleName");
dAdapt.InsertCommand.Parameters.Add("@inLastName", AllDbType.String, 50, "LastName");
dAdapt.InsertCommand.Parameters.Add("@inEMail", AllDbType.String, 150, "EMail");
```

The UpdateCommand is the SQL command that will be executed to update rows, in the data source, that have been modified within the DataSet. The code does not contain anything new:

```
dAdapt.UpdateCommand = new AllDbCommand("UPDATE Authors SET FirstName = @inFirstName, “ +
       “MiddleName = @inMiddleName, LastName = @inLastName, EMail = @inEMail " +
       "WHERE AuthorID = @inAuthorID", connect);

dAdapt.UpdateCommand.Parameters.Add("@inFirstName", AllDbType.String, 50, "FirstName");
dAdapt.UpdateCommand.Parameters.Add("@inMiddleName", AllDbType.String, 50, "MiddleName");
dAdapt.UpdateCommand.Parameters.Add("@inLastName", AllDbType.String, 50, "LastName");
dAdapt.UpdateCommand.Parameters.Add("@inEmail", AllDbType.String, 150, "Email");
dAdapt.UpdateCommand.Parameters.Add("@inAuthorID", AllDbType.Int32, 4, "AuthorID");
```

In the preceding WHERE clause, we use the key AuthorID, which is an auto-generated column that can't be changed, to find the row to update. This simplifies things because, if the key used to find the row to update can be changed during the update process, then when it's changed, the WHERE clause won't be able to find the right row due to the changed key not matching the original key in the database.

So, are you stuck with only being able to use unchangeable keys? Fortunately, the answer is no. When changed, DataRows store their original values so that they can be accessed for this exact reason (they can be used for rolling back changes as well). Let's pretend you can update AuthorID. Here is the code that needs to be changed:

```
dAdapt.UpdateCommand = new AllDbCommand("UPDATE Authors SET FirstName = @inFirstName, “ +
       “MiddleName = @inMiddleName, LastName = @inLastName, EMail = @inEMail " +
       "AuthorID = @inAuthorID " +
       "WHERE AuthorID = @inOldAuthorID", connect);

// ... all the parameters plus

dAdapt.UpdateCommand.Parameters.Add("@inOldAuthorID", AllDbType.Int32, 4,
       "AuthorID").SourceVersion = DataRowVersion.Original;
```

The DeleteCommand is the SQL command that will be executed when a DataRow is removed from the DataSet, which needs to be deleted now from the data source. Nothing new to explore here in the code:

```
dAdapt.DeleteCommand = new AllDbCommand("DELETE FROM Authors WHERE AuthorID = @inAuthorID", connect);

dAdapt.DeleteCommand.Parameters.Add("@inAuthorID", AllDbType.Int32, 4, "AuthorID");
```

You create a DataSet with a simple constructor. To fill the DataSet, you call the AllDbDataAdapter class's Fill() method. The Fill() method takes two parameters: a pointer to the DataSet and the name of the data source table that you will be filling the DataSet with.

```
dSet = new DataSet();
dAdapt.Fill(dSet, "Authors");
```

Selecting Rows

You have many ways of selecting records from the DataSet. A common way of getting all the rows from a table is to use the DataRow collection found in the Rows property of the table. This collect can be walked through using the foreach statement. You populate the list authors box doing exactly that.

One issue you need to address when selecting rows is that some databases don't support the autoincrement property. Fortunately, the dataset and the database don't have to be identical (at least in this regard). So, to allow auto incrementing to occur, you can add the autoincrement property to a column. But, this is not enough, as you will get a duplicate key exception if you try to add a column that auto-increments to an existing value. To fix this, we keep track of the highest key value and then seed the autoincrement with that value plus one.

```
DataTable dt = dSet.Tables["Authors"];

if (dt == null)
      throw new Exception("No Authors Table");

int lastID = 1;

foreach (DataRow row in dt.Rows)
{
      lbAuthors.Items.Add(new ListBoxItem(row));

      lastID = Convert.ToInt32(row["AuthorID"]);
}

if (dt.Columns["AuthorID"].AutoIncrement == false)
{
      dt.Columns["AuthorID"].AutoIncrement = true;
      dt.Columns["AuthorID"].AutoIncrementSeed = lastID+1;
}
```

One thing that is often overlooked about the list box is that it does not need to be a string value passed to it. We use this feature by passing it a class value that contains the DataRow itself. We then use the ToString() override to display the appropriate value. Now, since a list box item contains the full DataRow, we have access later to all the columns that make up the row and not just the one we want to display.

```
private class ListBoxItem

{
    public DataRow row;

    public ListBoxItem(DataRow row)
    {
        this.row = row;
    }

    public override string ToString()
    {
        return string.Format("{0} - {1} {2} {3} - {4}",
            row["AuthorID"],
            row["FirstName"],
            row["MiddleName"],
            row["LastName"],
            row["Email"]);
    }
}
```

Inserting Rows

Inserting a new row or, in this case, a new author, is done by updating the text boxes with the information about the author and then clicking the Add button.

A good portion of the following code is validating, updating the list box, and cleaning up for text boxes. The actual ADO.NET-related code simply creates a new row, updates the columns with the information in the list boxes, and adds the row to the DataTable.

Notice that the actual insertion of the row into the data source with the Update() method is not found in this method. The reason for this is that we want to be able to commit or roll back all changes at one time using the Commit and Rollback buttons. Thus, the Update() method only occurs in the Commit button event. When the Update() method finally gets called, the UpdateCommand (which was coded previously) will get executed:

```
private void bnAdd_Click(object sender, System.EventArgs e)
{
    bool Cancel = false;

    errorProvider.SetError(tbFirstName, "");
    errorProvider.SetError(tbLastName, "");

    if (tbFirstName.Text.Trim().Length <= 0)
    {
        errorProvider.SetError(tbFirstName, "First Name can not be blank");
        Cancel = true;
    }

    if (tbLastName.Text.Trim().Length <= 0)
    {
        errorProvider.SetError(tbLastName, "Last Name can not be blank");
        Cancel = true;
    }

    if (!Cancel)
    {
        // Create a new row in the DataTable
        DataTable dt = dSet.Tables["Authors"];
        DataRow row = dt.NewRow();

        // Update the columns with the new author information
        row["FirstName"] = tbFirstName.Text;
        row["MiddleName"]  = tbMiddleName.Text;
        row["LastName"]  = tbLastName.Text;
        row["Email"]  = tbEmail.Text;

        // Add the row to the Rows collection
        dt.Rows.Add(row);

        // Add the new row to the list box
        lbAuthors.Items.Add(new ListBoxItem(row));

        // blank out the text boxes
        tbFirstName.Text = "";
        tbMiddleName.Text = "";
        tbLastName.Text = "";
        tbEmail.Text = "";
    }
}
```

Updating Rows

Updating an author row is handled when you select a row out of the list box, update the text boxes, and finally click the Update button.

The ADO.NET-related code to update the author requires that you first select the row to be updated using the DataTable class's Select() method. Once you have the row, you update the author information in the row columns. Like when you added a row, the Update() method does not get called until the Commit button is clicked, but when the Update() method finally gets called, the UpdateCommand ends up being executed.

One break from standard ADO.NET coding here is the use of the AllDbControl's MakeTableWritable static helper method. We use it here because not all databases return writable DataRows. Because of this, when you try to assign new values to the row, an exception is thrown. Adding this method fixes this issue.

```
private void bnUpdate_Click(object sender, System.EventArgs e)
{
    // make sure we have a selected author from the listbox
    if (lbAuthors.SelectedItem == null)
        return;

    errorProvider.SetError(tbFirstName, "");
    errorProvider.SetError(tbLastName, "");

    bool Cancel = false;
    if (tbFirstName.Text.Trim().Length <= 0)
    {
        errorProvider.SetError(tbFirstName, "First Name can not be blank");
        Cancel = true;
    }

    if (tbLastName.Text.Trim().Length <= 0)
    {
        errorProvider.SetError(tbLastName, "Last Name can not be blank");
        Cancel = true;
    }

    if (!Cancel)
    {
        DataTable dt = dSet.Tables["Authors"];
        DataRow[] row =
            dt.Select(string.Format("AuthorID={0}",
                        ((ListBoxItem)lbAuthors.SelectedItem).row["AuthorID"]));

        AllDbControl.MakeTableWritable(dt); // Some databases retreive column as Readonly

        // Since we know that AuthorID is unique only one row will be returned
        // Update the row with the text box information
        row[0]["FirstName"] = tbFirstName.Text;
        row[0]["MiddleName"]  = tbMiddleName.Text;
        row[0]["LastName"]  = tbLastName.Text;
        row[0]["Email"]  = tbEmail.Text;

        // Update listbox
        int index = lbAuthors.SelectedIndex;
        lbAuthors.Items.Insert(lbAuthors.SelectedIndex, new ListBoxItem(row[0]));
        lbAuthors.Items.RemoveAt(lbAuthors.SelectedIndex);
        lbAuthors.SelectedIndex = index;
    }
}
```

Deleting Rows

Deletion of an author DataRow happens when you click a row in the list box and then click the Delete button.

The code to handle deleting a row is a little tricky, as it requires the use of transactional access to the DataSet. First, you need to select the row. Then you call its Delete() method. Deleting a record in the DataSet does not actually occur until the change in accepted. At this point only, a flag is set in the DataRow.

Also, like inserting and updating, the actual updating of the database does not occur until the Update() method is called when the Commit button is clicked. Ultimately, when the Update method is called, the DeleteCommand (built previously) will be executed:

```
private void bnDelete_Click(object sender, System.EventArgs e)
{
    // make sure we have a selected author from the listbox
    if (lbAuthors.SelectedItem == null)
        return;

    DataTable dt = dSet.Tables["Authors"];
    DataRow[] row =
        dt.Select(string.Format("AuthorID={0}",
        ((ListBoxItem)lbAuthors.SelectedItem).row["AuthorID"]));

    // Since we know that AuthorID is unique only one row will be returned
    // Delete the row
    row[0].Delete();

    // all went well, delete the row from list box
    lbAuthors.Items.RemoveAt(lbAuthors.SelectedIndex);

    // blank out the text boxes
    tbFirstName.Text = "";
    tbMiddleName.Text = "";
    tbLastName.Text = "";
    tbEmail.Text = "";

    lbAuthors.SelectedIndex = -1;
}
```

Committing and Rolling Back Changed Rows

Because a DataSet is disconnected from the database, anything that you do to it will not get reflected in the actual database until you force an update using the Update() method. Because this is the case, it is really a simple matter to either commit or roll back any changes that you have made to the DataSet.

To commit the changes to the database to click the commit button, this simply calls the Update() method, which will walk through the DataSet and update any changed records in its corresponding database record. Depending on the type of change, the appropriate SQL command (insert, update, or delete) will be executed.

To commit the changes to the DataSet, you need to call the DataSet's AcceptChanges() method, which will cause the DataSet to accept all changes that were made to it.

```
private void bnCommit_Click(object sender, System.EventArgs e)
{
    dAdapt.Update(dSet, "Authors");
    dSet.AcceptChanges();

    lbAuthors.Items.Clear();

    dSet = new DataSet();
    dAdapt.Fill(dSet, "Authors");

    DataTable dt = dSet.Tables["Authors"];

    if (dt == null)
        throw new Exception("No Authors Table");

    int lastID = 1;

    foreach (DataRow row in dt.Rows)
    {
        lbAuthors.Items.Add(new ListBoxItem(row));

        lastID = Convert.ToInt32(row["AuthorID"]);
    }

    if (dt.Columns["AuthorID"].AutoIncrement == false)
    {
        dt.Columns["AuthorID"].AutoIncrement = true;
        dt.Columns["AuthorID"].AutoIncrementSeed = lastID+1;
    }

    // blank out the text boxes
    tbFirstName.Text = "";
    tbMiddleName.Text = "";
    tbLastName.Text = "";
    tbEmail.Text = "";

    lbAuthors.SelectedIndex = -1;
}
```

To roll back any changes, simply don't call the Update() method, and call the DataSet's RejectChanges() method to delete all changes in the DataSet that you have made since you last committed:

```
private void bnRollback_Click(object sender, System.EventArgs e)
{
    dSet.RejectChanges();

    lbAuthors.Items.Clear();
    dSet = new DataSet();
    dAdapt.Fill(dSet, "Authors");

    DataTable dt = dSet.Tables["Authors"];

    if (dt == null)
        throw new Exception("No Authors Table");

    int lastID = 1;

    foreach (DataRow row in dt.Rows)
    {
        lbAuthors.Items.Add(new ListBoxItem(row));

        lastID = Convert.ToInt32(row["AuthorID"]);
    }

    if (dt.Columns["AuthorID"].AutoIncrement == false)
    {
        dt.Columns["AuthorID"].AutoIncrement = true;
        dt.Columns["AuthorID"].AutoIncrementSeed = lastID+1;
    }

    // blank out the text boxes
    tbFirstName.Text = "";
    tbMiddleName.Text = "";
    tbLastName.Text = "";
    tbEmail.Text = "";

    lbAuthors.SelectedIndex = -1;
}
```

Summary

In this document, we covered a large portion of the AllDb class library and well as some of the database generic aspects of ADO.NET. We started out by covering the basics of AllDb. We then moved on to the building of an AllDb compatible database and some of the more prevalent gotchas. Next, we covered how to connect, query, insert, update, delete, count, and sum rows of a database using AllDb's connected access to the database. Finally, we learned how to do the same things with AllDb's disconnected access, in the process building a simple Win Form author maintenance tool.

You have now learned the code you will need to implement AllDb in either a connected or disconnected manner. The world of databases is now open to you when you create your applications.