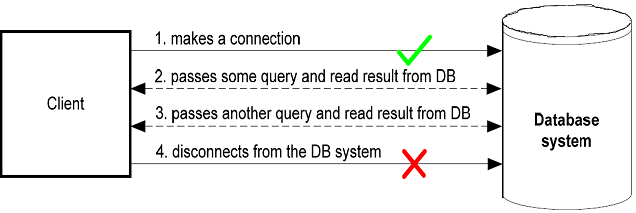
**Chapter Four**

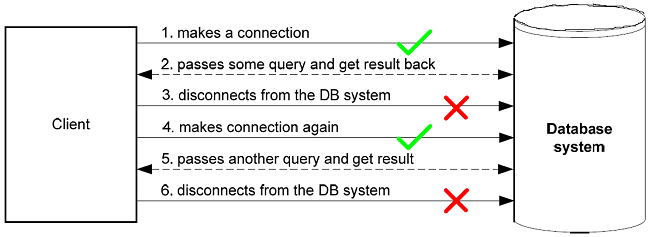
**Introduction to ADO.NET**

What is ADO.NET?

* In a simple term, you can think of ADO.NET, as a set of classes (framework) that can be used to interact with data sources like databases and XML files.
* This data can, and then is consumed in any .NET application.
* ADO stands for Microsoft Activex Data Object.
* The following are a few of the different types of .NET applications that use ADO.NET to connect to database, execute commands, and retrieve data.
  + ASP.NET Web Applications
  + Windows Applications
  + Console Applications
* ADO.NET is the primary data access API for the .NET Framework.
* We usually interact with database systems through SQL queries or stored procedures.
* The best thing about ADO.Net is that it is extremely flexible and efficient.
* ADO.Net also introduces the concept of disconnected data architecture.
* In traditional data access components, you made a connection to the database system and then interacted with it through SQL queries using the connection.
* The application stays connected to the DB system even when it is not using DB services.
* This commonly wastes valuable and expensive database resources, as most of the time applications only query and view the persistent data.
* ADO.Net solves this problem by managing a local buffer of persistent data called a data set.
* Your application automatically connects to the database server when it needs to run a query and then disconnects immediately after getting the result back and storing it in the dataset.
* This design of ADO.Net is called disconnected data architecture
* It should be noted that ADO.Net also provides connection oriented traditional data access services.

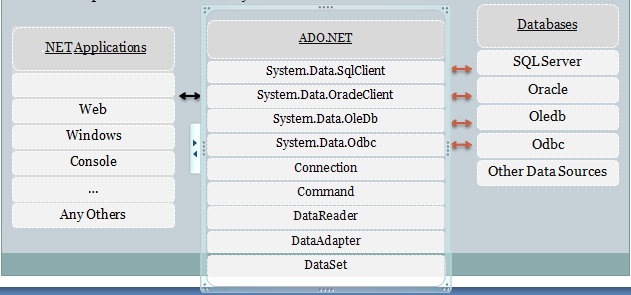


**Traditional data access architecture**



ADO.Net disconnected data access architecture

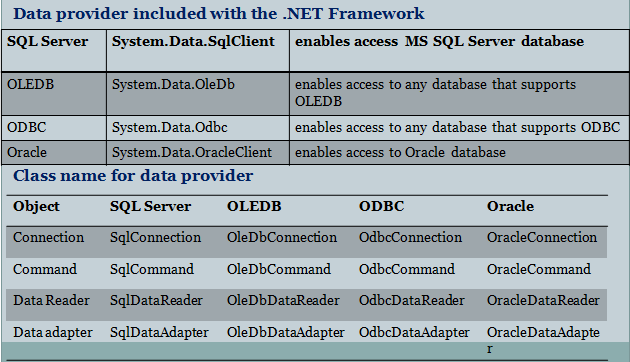
* Another important aspect of disconnected architecture is that it maintains a local repository of data in the dataset object.
* The dataset object stores the tables, their relationship and their different constraints.
* The user can perform operations like update, insert and delete on this dataset locally, and the changes made to the dataset are applied to the actual database as a batch when needed.
* This greatly reduces network traffic and results in better performance.
* .NET Data Providers:
  + Data provider for SQL Server - System.Data.SqlClient.
  + Data provider for Oracle - System.Data.OracleClient.
  + Data provider for OLEDB - System.Data.OleDb.
  + Data provider for ODBC - System.Data.Odbc.

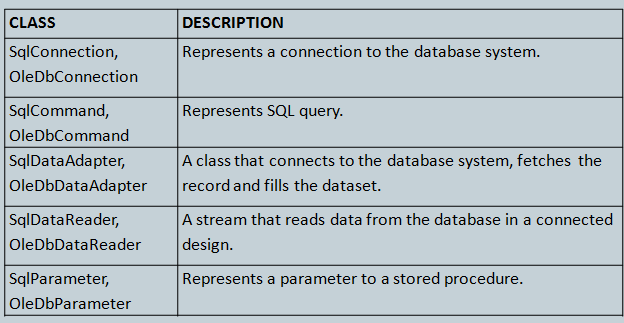


**Different components of ADO.Net**

* It provides the classes that you use as you develop database application with C# and other .NET languages.
* These classes can be divided into two
  + **The .NET data provider** which provide the classes that you use to access the data in database
  + **Dataset** which provide the classes that you can use to store and work with data in your application
* **The .NET data providers**
  + A .NET data provider is a set of classes that enables you to access data that’s managed by a particular database server.
  + All .NET data providers must include core classes for creating the four types of objects.
    - **Connection** establishes a connection to a database server
    - **Command** represents an individual SQL statement that can be executed against the database
    - **Data** **reader** provides read-only forward access to the data in a database
    - **Data** **adapter** provides the link between the command and connection objects and a dataset object
* All generic classes for data access are contained in the System.Data namespace.

|  |  |
| --- | --- |
| **CLASS** | **DESCRIPTION** |
| DataSet | The DataSet is a local buffer of tables or a collection of disconnected record sets. |
| DataTable | A DataTable is used to contain data in tabular form using rows and columns. |
| DataRow | Represents a single record or row in a DataTable. |
| DataColumn | Represents a column or field of a DataTable. |
| DataRelation | Represents the relationship between different tables in a data set.. |
| Constraints | Represents the constraints or limitations that apply to a particular field or column. |

****

****

* DataSet provides a disconnected representation of result sets from the data Source, and it is completely independent from the Data Source.
* DataSet provides much greater flexibility when dealing with related Result Sets.
* Dataset is structured much like a relational database.
* It can contain one or more tables, and each table can contain one or more columns and rows.
* You can also work with the data in a database without using a data adapter.
* Use command and connection objects to access database.
* Instead of using a data adapter to execute the commands, you can execute the commands directly.
* When you do that, you also have to provide code to handle the result of the command.
* If you issue a command that contains an Insert, Update, or Delete statement the result is an integer that indicates the number of rows that were affected by the operation.
* You can use that operation to determine if the operation was successful.
* If you execute a command that contains a Select statement, the result is a result set that contains the rows you requested.
* To read through the rows in the result set, use a data reader object.
* The steps for executing database command in ADO.NET are:
  + Create connection object
  + Open the connection
  + Create a command object encapsulating both Sql command and the connection
  + Call a method on the command object to execute the command
  + Close the connection by calling the close on the connection object

**Performing common data access tasks with ADO.Net**

We will use the MS SQL server and MS Access database systems to perform the data access tasks. SQL Server is used because probably most of the time you will be using MS SQL server when developing .Net applications.

For SQL server, we will be using classes from the System.Data.SqlClient namespace. Access is used to demonstrate the OleDb databases. For Access we will be using classes from the System.Data.OleDb namespace. In fact, there is nothing different in these two approaches for developers and only two or three statements will be different in both cases. We will highlight the specific statements for these two using comments like:

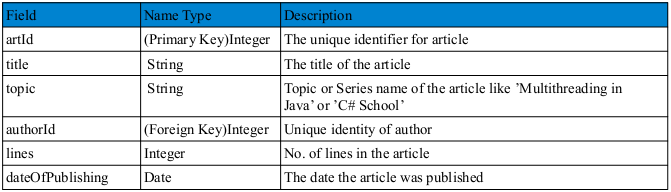
// For SQL server

SqlDataAdapter dataAdapter = new SqlDataAdapter (commandString, conn);

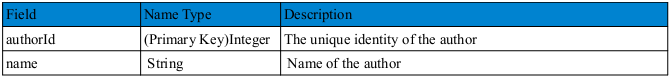
// For Access

OleDbDataAdapter dataAdapter = new OleDbDataAdapter (commandString, conn);

For the example code, we will be using a database named ’ProgrammersHeaven’. The database will have a table named ’Article’. The fields of the table ’Article’ are:



The ’ProgrammersHeaven’ database also contains a table named ’Author’ with the following fields:



**Accessing Data using ADO.Net**

Data access using ADO.Net involves the following steps:

* Defining the connection string for the database server
* Defining the connection (SqlConnection or OleDbConnection) to the database using the connection string
* Defining the command (SqlCommand or OleDbCommand) or command string that contains the query
* Defining the data adapter (SqlDataAdapter or OleDbDataAdapter) using the command string and the connection object
* Creating a new DataSet object
* If the command is SELECT, filling the dataset object with the result of the query through the data adapter
* Reading the records from the DataTables in the datasets using the DataRow and DataColumn objects
* If the command is UPDATE, INSERT or DELETE, then updating the dataset through the data adapter
  + Accepting to save the changes in the dataset to the database

Since we are demonstrating an application that uses both SQL Server and Access databases we need to include the following namespaces in our application:

using System.Data;

using System.Data.OleDb; // for Access database

using System.Data.SqlClient; // for SQL Server

Let’s now discuss each of the above steps individually

**Defining the connection string**

The connection string defines which database server you are using, where it resides, your user name and password and optionally the database name.

For SQL Server, we have written the following connection string:

// for Sql Server

string connectionString = "server=P-III; database=programmersheaven;uid=sa; pwd=;";

First of all we have defined the instance name of the server, which is P-III on my system. Next we defined the name of the database, user id (uid) and password (pwd). Since my SQL server doesn’t have a password for the System Administrator (sa) user, I have left it blank in the connection string. (Yes I know this is very dangerous and is really a bad practice - never, ever use a blank password on a system that is accessible over a network)

For Access, we have written the following connection string:

// for MS Access

string connectionString = "provider=Microsoft.Jet.OLEDB.4.0;data source = c:\\programmersheaven.mdb";

First we have defined the provider of the access database. Then we have defined the data source which is the address of the target database.

**Defining a Connection**

A connection is defined using the connection string. This object is used by the data adapter to connect to and disconnect from the database For SQL Server the connection is created like this:

// for Sql Server

SqlConnection conn = new SqlConnection (connectionString);

And for Access the connection is created like this:

// for MS Access

OleDbConnection conn = new OleDbConnection (connectionString);

Here we have passed the connection string to the constructor of the connection object.

**Defining the command or command string**

The command contains the query to be passed to the database. Here we are using a command string. We will see the command object (SqlCommand or OleDbCommand) later in the lesson. The command string we have used in our application is:

string commandString = "SELECT " +

"artId, title, topic, " +

"article.authorId as authorId, " +

"name, lines, dateOfPublishing " +

"FROM " +

"article, author " +

"WHERE " +

"author.authorId = article.authorId";

Here we have passed a query to select all the articles along with the author’s name. Of course you may use a simpler query, such as:

string commandString = "SELECT \* from article";

* Some important built in methods uses in the Command Object to execute the SQL statements

|  |  |
| --- | --- |
| **ExecuteNonQuery** | **used for executing statements that do not return result sets (i.e. statements like insert data, update data etc.)** |
| **ExecuteReader** | **to perform database queries and obtain the results Returns a DataReader object DataReader has methods and properties that you can call to iterate over the result set,**  **such as GetValue, GetName, Read, GetDecimal, …** |
| **ExecuteScalar** | **to retrieve a single value from Database after the execution of the SQL Statement returns the first row of the first column in the result set**  **used to execute SQL functions such as COUNT, AVG, MIN, MAX, and SUM which**  **returns a single row single, column result set** |

* **The following example uses a SqlCommand object to delete a record from the LibrarySys database's “Book” table using an SQL DELETE command**

SqlConnection conn = new SqlConnection ("server= localhost; database=LibrarySys; uid=sa; pwd=");

try {

conn.Open ();

SqlCommand cmd = new SqlCommand ();

cmd.CommandText = "delete from Book where ISBN = 'BU1032'";

cmd.Connection = conn;

cmd.ExecuteNonQuery (); // Execute the command

}

catch (SqlException ex) {

// TODO: Handle the exception

}

finally{

conn.Close ();

}

* You can make your code more concise by creating a SqlCommand object and initializing its Connection and CommandText properties in one step:

SqlConnection conn = new SqlConnection ("server= localhost; database= pubs; uid=sa; pwd=");

try {

conn.Open ();

SqlCommand cmd = new SqlCommand("delete from titles where title\_id= 'BU1032'", conn);

cmd.ExecuteNonQuery (); // Execute the command

}

catch (SqlException ex) {

// TODO: Handle the exception

}

finally

{

conn.Close ();

}

* Here’s an example that uses ExecuteNonQuery to add a record to the LibrarySys database's “Book” table using an INSERT command:

SqlConnection conn = new SqlConnection ("server= localhost; database= LibrarySys; uid= sa; pwd=");

try {

conn.Open ();

SqlCommand cmd = new SqlCommand

("insert into Book (ISBN, title, author, price) " +

"values ('DBU1001', 'Programming Microsoft .NET', " +

“ ‘Abebe’, 45)", conn);

cmd.ExecuteNonQuery ();

}

catch (SqlException ex) {

// TODO: Handle the exception

}

finally{

conn.Close ();

}

* The following example writes the largest Book price recorded in the LibrarySys database to a TextBox:

SqlConnection conn = new SqlConnection

("server= localhost; database= LibrarySys; uid= sa; pwd=");

try

{

conn.Open ();

SqlCommand cmd = new SqlCommand

("select max (price) from Book", conn);

decimal amount = (decimal) cmd.ExecuteScalar();//cast object to decimal

txtprice.Text = amount.ToString(“C”);

}

catch (SqlException ex) {

//Todo: Handle exception

}

Finally {

conn.Close ();

}

* ExecuteScalar is used to retrieve BLOBs (binary large objects) from databases.
* The following example retrieves an image from the “photo” field of the LibrarySys database's “Customer” table and encapsulates it in a bitmap:

MemoryStream stream = new MemoryStream ();

SqlConnection conn = new SqlConnection("server= localhost; database= LibrarySys; uid= sa; pwd=");

try {

conn.Open ();

SqlCommand cmd=new SqlCommand ("select photo from customer where custId='DBU1001'", conn);

byte[] blob = (byte[]) cmd.ExecuteScalar ();

stream.Write (blob, 0, blob.Length);

Bitmap bitmap = new Bitmap (stream);

// TODO: Use the bitmap

pictureBox1.Image = bitmap;

bitmap.Dispose ();

}

catch (SqlException ex) {

// TODO: Handle the exception

}

Finally {

stream.Close ();

conn.Close ();

}

* Once the bitmap is created, you can do whatever you want with it: display it in a Windows form, stream it back. In an HTTP response, or whatever. Note that in order for this sample to compile, you must include using statements that import the System.IO and System.Drawing namespaces as well as System and System.Data.SqlClient.
* You might be interested in knowing how to write BLOBs to databases, too.
* The secret is to call ExecuteNonQuery on a command object that wraps an INSERT command containing an input parameter whose type is byte[]. To demonstrate, the following example inserts a record into the LibrarySys database's “Customer” table and includes a BLOB in the record's “photo” field:

FileStream stream = new FileStream ("photo.jpg", FileMode.Open);

byte[] blob = new byte[ stream.Length ];

stream.Read (blob, 0, (int) stream.Length);

stream.Close ();

SqlConnection conn = new SqlConnection

(""Data Source= CoolTech\ sqlexpress; Initial Catalog= LibrarySys; Integrated

Security=True" =");

Try {

conn.Open ();

SqlCommand cmd = new SqlCommand

("insert into customer (CustId, photo) values ('DBU1001', @photo)",

conn);

cmd.Parameters.Add ("@photo", blob);

cmd.ExecuteNonQuery ();

}

catch (SqlException ex) { // TODO: Handle the exception }

Finally { conn.Close (); }

* The following example uses ExecuteReader and the resultant SqlDataReader to write the titles of all the books listed in the LibrarySys database to a ListBox:

SqlConnection conn = new SqlConnection

("server= localhost; database=LibrarySys; uid= sa; pwd=");

try

{

conn.Open ();

SqlCommand cmd = new SqlCommand ("select \* from Book", conn);

SqlDataReader reader = cmd.ExecuteReader();

while (reader.Read ())

listBox1.Items.Add(reader["Title"].ToString());

}

catch (SqlException ex) {

Console.WriteLine (ex.Message);

}

finally {

conn.Close ();

}

* Each call to SqlDataReader.Read returns one row from the result set. This example uses a property indexer to extract the value of the record's “Title” field. Fields can be referenced by name or by numeric index (0-based, of course).

**Defining the Data Adapter**

Next we need to define the data adapter (SqlDataAdapter or OleDbDataAdapter). The data adapter stores your command (query) and connection, and using these connects to the database when asked, fetches the result of the query and stores it in the local dataset.

For SQL Server, the data adapter is created like this:

// for Sql Server

SqlDataAdapter dataAdapter = new SqlDataAdapter (commandString, conn);

And for Access, the data adapter is created like this:

// for MS Access

OleDbDataAdapter dataAdapter = new OleDbDataAdapter (commandString, conn);

Here we have created a new instance of the data adapter and supplied it the command string and connection object in the constructor call.

**Creating and filling the DataSet**

Finally we need to create an instance of the DataSet. As we mentioned earlier, a DataSet is a local & offline container of data. The DataSet object is created simply:

DataSet ds = new DataSet ();

Now we need to fill the DataSet with the result of the query. We will use the dataAdapter object for this purpose and call its Fill () method. This is the step where the data adapter connects to the physical database and fetches the result of the query.

dataAdapter.Fill(ds, "prog");

Here we have called the Fill() method of dataAdapter object. We have supplied it the dataset to fill and the name of the table (DataTable) in which the result of query is filled.

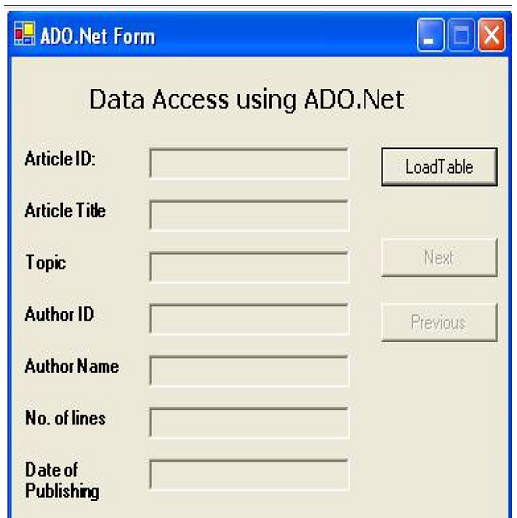
This is all we need to connect and fetch data from the database. Now the result of the query is stored in the dataset object in the prog table, which is an instance of the DataTable. We can get a reference to this table by using the indexer property of the dataset object’s Tables collection.

DataTable dataTable = ds.Tables ["prog"];

The indexer we have used takes the name of the table in the dataset and returns the corresponding DataTable object. Now we can use the tables Rows and Columns collections to access the data in the table.

**A Demonstration Application**

Let’s now create a demonstration application for accessing data. First create a windows form application and make the layout shown the following snapshot.



We have set the Name property of the text boxes (from top to bottom) as txtArticleID, txtArticleTitle, txtArticleTopic, txtAuthorId, txtAuthorName, txtNumOfLines and txtDateOfPublishing. Also we have set the ReadOnly property of all the text boxes to true as don’t want the user to change the text. The names of the buttons (from top to bottom) are btnLoadTable, btnNext and btnPrevious. Initially we have disabled the Next and Previous buttons (by setting their Enabled property to false).

We have also defined three variables in the Form class:

public class ADOForm : System.Windows.Forms.Form

{

DataTable dataTable;

int currRec=0;

int totalRec=0;

The dataTable object will be used to reference the table returned as a result of the query. The currRec and totalRec integer variables are used to keep track of the current record and total number of records in the table.

**Loading tables**

For the LoadTable button, we have written the following event handler

private void btnLoadTable\_Click(object sender, System.EventArgs e)

{

// for Sql Server

string connectionString = "server=P-III; database=programmersheaven;uid=sa; pwd=;";

// for MS Access

/\*string connectionString = "provider=Microsoft.Jet.OLEDB.4.0;" +

"data source = c:\\programmersheaven.mdb";\*/

// for Sql Server

SqlConnection conn = new SqlConnection(connectionString);

// for MS Access

//OleDbConnection conn = new OleDbConnection(connectionString);

string commandString = "SELECT " +

"artId, title, topic, " +

"article.authorId as authorId, " +

"name, lines, dateOfPublishing " +

"FROM " +

"article, author " +

"WHERE " +

"author.authorId = article.authorId";

// for Sql Server

SqlDataAdapter dataAdapter = new SqlDataAdapter(commandString, conn);

// for MS Access

//OleDbDataAdapter dataAdapter = new OleDbDataAdapter(commandString, conn);

DataSet ds = new DataSet();

dataAdapter.Fill(ds, "prog");

dataTable = ds.Tables["prog"];

currRec = 0;

totalRec = dataTable.Rows.Count;

FillControls();

btnNext.Enabled = true;

btnPrevious.Enabled = true;

}

First we created the connection, data adapter and filled the dataset object, all of which we have discussed earlier. It should be noted that we have commented out the code for the OleDb provider (MS-Access) and are using SQL Server specific code. If you would like to use an Access databases, you can simply comment the SQL server code out and de-comment the Access code.

Next, we have assigned the data table resulting from the query to the dataTable object which we declared at the class level, assigned zero to currRec variable and assigned the number of rows in the dataTable to the totalRec variable:

dataTable = ds.Tables["prog"];

currRec = 0;

totalRec = dataTable.Rows.Count;

Then we called the FillControls () method, which fills the controls (text boxes) on the form with the current record of the table "prog". Finally we enabled the Next and Previous Buttons.

Filling the controls on the Form The FillControls() method in our program fills the controls on the form with the current record of the data table.

The method is defined as follows:

private void FillControls()

{

txtArticleId.Text = dataTable.Rows[currRec]["artId"].ToString();

txtArticleTitle.Text = dataTable.Rows[currRec]["title"].ToString();

txtArticleTopic.Text = dataTable.Rows[currRec]["topic"].ToString();

txtAuthorId.Text = dataTable.Rows[currRec]["authorId"].ToString();

txtAuthorName.Text = dataTable.Rows[currRec]["name"].ToString();

txtNumOfLines.Text = dataTable.Rows[currRec]["lines"].ToString();

txtDateOfPublishing.Text = dataTable.Rows[currRec]["dateOfPublishing"].ToString();

}

Here we have set the Text property of the text boxes to the string values of the corresponding fields of the current record. We have used the Rows collection of the dataTable and using its indexer we have got the DataRow representing the current record. We have then accessed the indexer property of this DataRow using the column name to get the data in the respective field. If this explanation looks weird to you, you can simplify the above statements to:-

DataRow row = dataTable.Rows[currRec]; // getting current row

object data = row["artId"]; // getting data in the artId field

string strData = data.ToString(); // converting to string

txtArticleId.Text = strData; // display in the text box

which is equivalent to

txtArticleId.Text = dataTable.Rows [currRec]["artId"].ToString();

**Navigating through the records**

Navigating through the records is again very easy. For the Next button, we have written the following simple event handler

private void btnNext\_Click(object sender, System.EventArgs e)

{

currRec++;

if(currRec>=totalRec)

currRec=0;

FillControls();

}

Here we first increment the integer variable currRec and check if it has crossed the last record (using the totalRec variable) in the table. If it has, then we move the current record to the first record. We then call the FillControls() method to display the current record on the form.

Similarly the event handler for the Previous button looks like this:

private void btnPrevious\_Click(object sender, System.EventArgs e)

{

currRec--;

if(currRec<0)

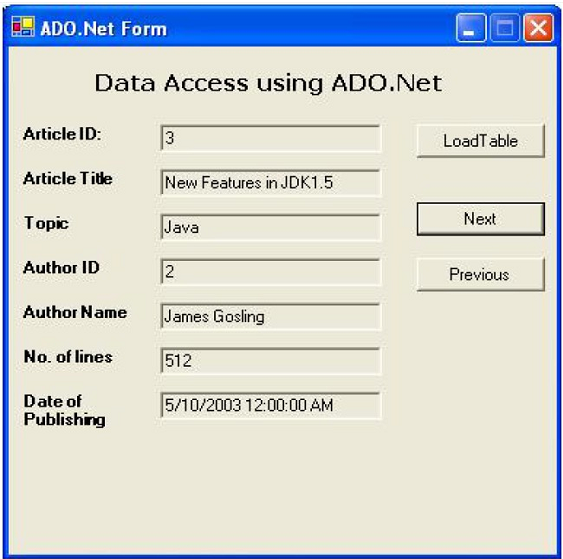
currRec=totalRec-1;

FillControls();

}

Here we decrement the currRec variable and check if has crossed the first record and if it has then we move it to the last record. Once again, we call the FillControls() method to display the current record.

Now you can navigate through the records using the Next and Previous buttons.



**Updating the table**

So far we have only selected the data from the database and haven’t changed any row, inserted new rows or deleted existing rows. Let’s learn how to perform these tasks one by one.

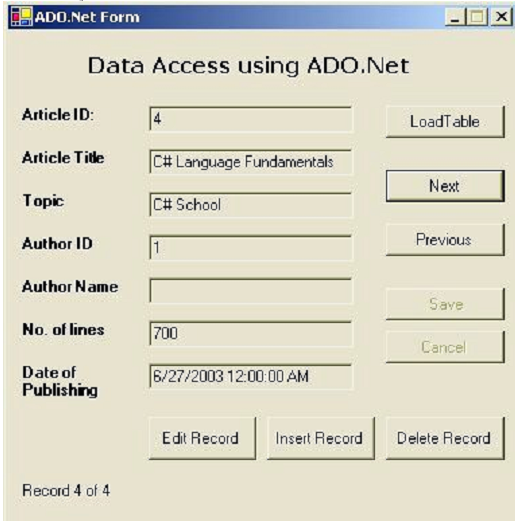
Please note that for this section, we will only use the "Article" table and will not be using the "Author" table, for the sake of simplicity. Updating a table in ADO.Net is very interesting and easy. You need to follow these steps to update, insert and delete records:

The Data Adapter class (SqlDataAdapter) has properties for each of the insert, update and delete commands.

First of all we need to prepare the command (SqlCommand) and add it to the data adapter object.

**Building the Application**

The application will finally look like this:



In this application, we have defined several data access objects (like SqlDataAdapter, DataSet) as private class members so that we can access them in different methods.

public class ADOForm : System.Windows.Forms.Form

{

// Private global members to be used in various methods

private SqlConnection conn;

private SqlDataAdapter dataAdapter;

private SqlCommandBuilder CB;//this object creates the delete, update and insert statements

//automatically

private DataTable dataTable;

private DataSet ds;

private int currRec=0;

private int totalRec=0;

private void btnLoadTable\_Click(object sender, System.EventArgs e)

{

this.Cursor = Cursors.WaitCursor;

string connectionString ="server=P-III; database=programmersheaven;uid=sa; pwd=;";

conn = new SqlConnection(connectionString);

string commandString = "SELECT \* from article";

dataAdapter = new SqlDataAdapter(commandString, conn);

CB=new SqlCommandBuilder(dataAdapter);//creates update commands for the dataAdapter

ds = new DataSet();

dataAdapter.Fill(ds, "article");

dataTable = ds.Tables["article"];

currRec = 0;

totalRec = dataTable.Rows.Count;

FillControls(); // show current record on the form

InitializeCommands(); // prepare commands

ToggleControls(true); // enable corresponding controls

this.Cursor = Cursors.Default;

}

We have changed the cursor to WaitCursor at the start of the method, and changed it to Default at the end of the method. Later we have called the InitializeCommands() method after filling the controls with the first record. The ToggleControls() method of our application

In the Load Table buttons event handler, we called the ToggleControls() method after initializing the commands

private void btnLoadTable\_Click(object sender, System.EventArgs e){

this.Cursor = Cursors.WaitCursor;

...

FillControls(); // show current record on the form

ToggleControls(true); // enable corresponding controls

this.Cursor = Cursors.Default;

}

We have defined the ToggleControls () method in our application to change the Enabled and ReadOnly properties of buttons and text boxes in our form at the respective times.

If the ToggleControls () method is called with a false boolean value it will take the form to the edit mode and if called with a true value it will take the form back to normal mode. The method is defined as:

private void ToggleControls(bool val)

{

txtArticleTitle.ReadOnly = val;

txtArticleTopic.ReadOnly = val;

txtAuthorId.ReadOnly = val;

txtNumOfLines.ReadOnly = val;

txtDateOfPublishing.ReadOnly = val;

btnLoadTable.Enabled = val;

btnNext.Enabled = val;

btnPrevious.Enabled = val;

btnEditRecord.Enabled = val;

btnInsertRecord.Enabled = val;

btnDeleteRecord.Enabled = val;

btnSave.Enabled = !val;

btnCancel.Enabled = !val;

}

**Editing (or Updating) Records**

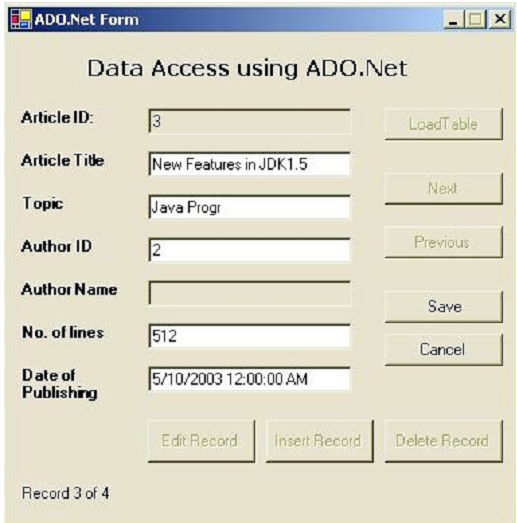
For editing the current record, we have provided an Edit Record button on the form. The event handler for this button is surprisingly very simple and is:

private void btnEditRecord\_Click(object sender, System.EventArgs e){

ToggleControls(false);

}

This event handler simply takes the form and the controls to edit mode by passing a false value to the ToggleControls() method presented above. When a user presses the Edit Record button, the form is changed, so it looks like:



As you can see now, the text boxes are editable and the Save and Cancel buttons are enabled. If the user wishes not to save the changes, they can select the Cancel button, and if they wishes to save the changes, they can select the Save button after making the changes.

private void btnSave\_Click(object sender, System.EventArgs e)

{

lblLabel.Text = "Saving Changes...";

this.Cursor = Cursors.WaitCursor;

DataRow row = dataTable.Rows[currRec];

row.BeginEdit();

row["title"] = txtArticleTitle.Text;

row["topic"] = txtArticleTopic.Text;

row["authorId"] = txtAuthorId.Text;

row["lines"] = txtNumOfLines.Text;

row["dateOfPublishing"] = txtDateOfPublishing.Text;

row.EndEdit();

dataAdapter.Update(ds, "article");

ds.AcceptChanges();

ToggleControls(true);

this.Cursor = Cursors.Default;

lblLabel.Text = "Changes Saved";

}

Here we first change the progress label (lblLabel) text to show the current progress and then change the cursor to wait cursor.

lblLabel.Text = "Saving Changes...";

this.Cursor = Cursors.WaitCursor;

Then we take a reference to the current record’s row and called its BeginEdit() method. The update on a row is usually bound by a DataRow’s BeginEdit() and EndEdit() methods. The BeginEdit() method temporarily suspends the events for the validation of row’s data. Within the BeginEdit() and EndEdit() boundary, we stored the changed values in the text boxes to the row.

DataRow row = dataTable.Rows[currRec];

row.BeginEdit();

row["title"] = txtArticleTitle.Text;

row["topic"] = txtArticleTopic.Text;

row["authorId"] = txtAuthorId.Text;

row["lines"] = txtNumOfLines.Text;

row["dateOfPublishing"] = txtDateOfPublishing.Text;

row.EndEdit();

After saving the changes in the row, we update the dataset and table by calling the Update method of data adapter.

This saves the changes in the local repository of data: dataset. To save the changed rows and tables to the physical database, we called the AcceptChanges() method of the DataSet class.

dataAdapter.Update(ds, "article");

ds.AcceptChanges();

Finally we brought the controls to normal mode by calling the ToggleControls() method and passing it the true value.

**Inserting Records**

To insert a record into the table, the user can select the Insert Record button. A record is inserted into the table by adding a new row to the DataTable’s Rows collection. Here is the event handler for the Insert Record button.

private void btnInsertRecord\_Click(object sender, System.EventArgs e)

{

DataRow row = dataTable.NewRow();

dataTable.Rows.Add(row);

totalRec = dataTable.Rows.Count;

currRec = totalRec-1;

row["artId"] = totalRec;

txtArticleId.Text = totalRec.ToString();

txtArticleTitle.Text = "";

txtArticleTopic.Text = "";

txtAuthorId.Text = "";

txtNumOfLines.Text = "";

txtDateOfPublishing.Text = DateTime.Now.Date.ToString();

ToggleControls(false);

}

First of all we set the insertSelected variable to true, so that later the Cancel button may get informed that the edit mode was set by the Insert Record button. We then created a new DataRow using the DataTable’s NewRow() method, added it to the Rows collection of the data table and updated the currRec and totalRec variables.

DataRow row = dataTable.NewRow();

dataTable.Rows.Add(row);

totalRec = dataTable.Rows.Count;

currRec = totalRec-1;

Now the new row is ready to have the new values inserted into it. Here we have set the artId field to the total number of records as we don’t want to allow the user to set the primary key field of the table. Of course this is just a design issue. You may want to allow your user to insert the primary key field value too.

row["artId"] = totalRec;

txtArticleId.Text = totalRec.ToString();

We then cleared all the text boxes, but filled the Date of Publishing text box with the current date in order to help the user, and finally set the edit mode by calling the ToggleControls() method.

txtArticleTitle.Text = "";

txtArticleTopic.Text = "";

txtAuthorId.Text = "";

txtNumOfLines.Text = "";

txtDateOfPublishing.Text = DateTime.Now.Date.ToString();

ToggleControls(false);

**Deleting a Record**

Deleting a record is again very simple. All you need to do is get a reference to the target row and call its delete() method. Then you need to call the Update() method of the data adapter and the AcceptChanges() method of the DataSet to permanently save your changes in the data table to the physical database. The event handler for the Delete Record button is:

private void btnDeleteRecord\_Click(object sender, System.EventArgs e)

{

DialogResult res = MessageBox.Show(

"Are you sure you want to delete the current record?",

"Confirm Record Deletion", MessageBoxButtons.YesNo);

if(res == DialogResult.Yes)

{

DataRow row = dataTable.Rows[currRec];

row.Delete();

dataAdapter.Update(ds, "article");

ds.AcceptChanges();

lblLabel.Text = "Record Deleted";

totalRec--;

currRec = totalRec-1;

FillControls();

}

Since selecting the delete record button will permanently delete the current record, we need to seek confirmation from the user to check they are sure they wish to go ahead with the deletion. For this purpose we present the user with a message box with Yes and No buttons.