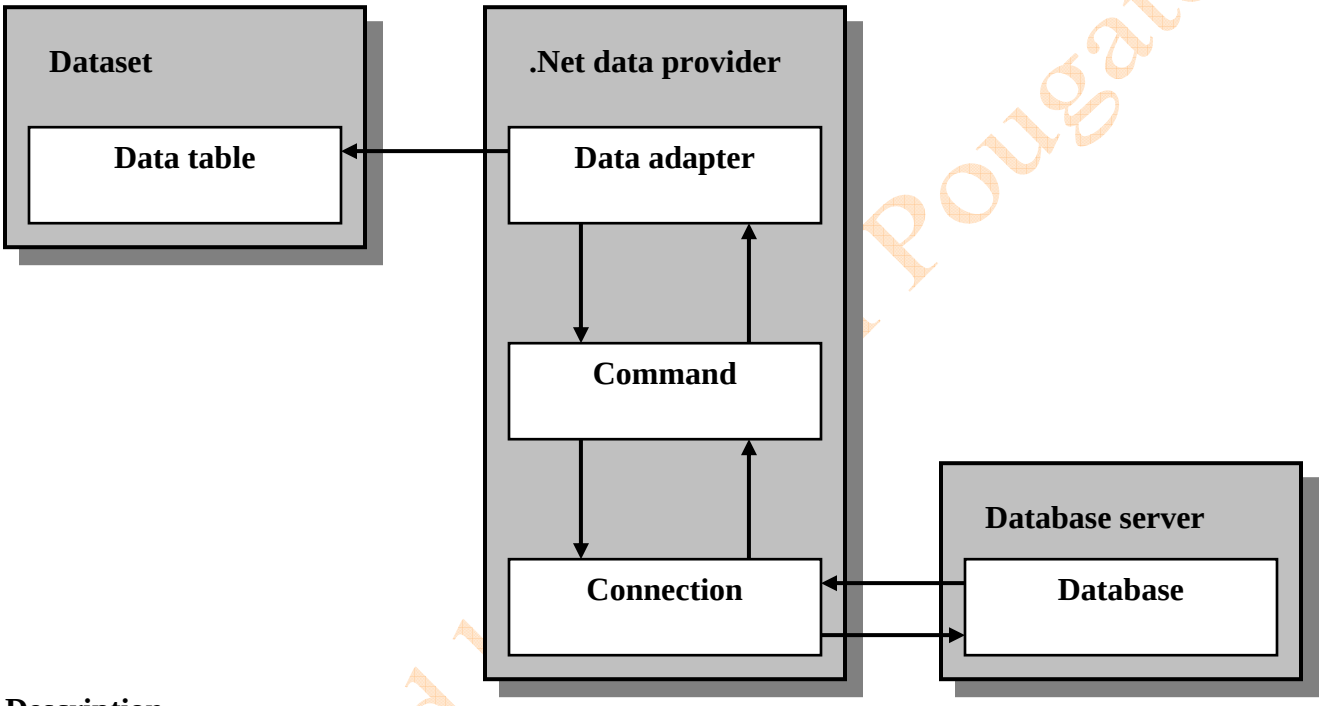
Lecture 8

ASP.NET database programming

ADO.NET 2.0

ADO.NET 2.0 (***Active Data Objects***) is the primary data access API for the .NET Framework. It provides the classes that are used when develop database applications.

Basic ADO.NET objects



Description

- ADO.NET uses two types of objects to access the data in a database: ***datasets***, which can contain one or more ***data tables***, and ***.NET data provider*** objects, which include ***data adapters, commands***, and ***connections*** data from
- A dataset stores data from the database so it can be accessed by the application. The .NET data provider objects retrieve data from and update data in the database.
- To retrieve data from a database and store it in a data table, a data adapter object issues a Select statement that's stored in a command object. Next, the command object uses a connection object to connect to the database and retrieve the data. Then, the data is passed back to the data adapter, which stores the data in the dataset.
- To update the data in a database based on the data in a data table, the data adapter object issues an Insert, Update, or Delete statement that's stored in a command object. Then, the command object uses a connection to connect to the database and update the data.

- The data provider remains connected to the database only long enough to retrieve or update the specified data. Then, it disconnects from the database and the application work with the data via the dataset object. This is referred to as ***disconnected data architecture***.
- All of the ADO.NET objects are implemented by classes in the ***System.Data*** namespace of the .NET Framework. However, the specific classes used to implement the connection, command, and data adapter objects depend on the .NET data provider you use.

Data Concurrency in ADO.Net

When multiple users attempt to modify data at the same time, controls need to be established in order to prevent one user's modifications from adversely affecting modifications from simultaneous users. The system of handling what happens in this situation is called ***concurrency control.***

Types of Concurrency Control

In general, there are three common ways to manage concurrency in a database:

- ***Pessimistic concurrency control*** - a row is unavailable to users from the time the record is fetched until it is updated in the database.
- ***Optimistic concurrency control*** - a row is unavailable to other users only while the data is actually being updated. The update examines the row in the database and determines whether any changes have been made. Attempting to update a record that has already been changed results in a concurrency violation.
- ***"Last in wins"*** - a row is unavailable to other users only while the data is actually being updated. However, no effort is made to compare updates against the original record; the record is simply written out, potentially overwriting any changes made by other users since you last refreshed the records.

Pessimistic Concurrency

Pessimistic concurrency is typically used for two reasons. First, in some situations there is high contention for the same records. The cost of placing locks on the data is less than the cost of rolling back changes when concurrency conflicts occur.

Pessimistic concurrency involves locking rows at the data source to prevent users from modifying data in a way that affects other users. In a pessimistic model, when a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the lock owner releases it.

Therefore, in a pessimistic currency model, a user who reads a row with the intention of changing it establishes a lock. Until the user has finished the update and released the lock, no one else can change that row. For this reason, pessimistic concurrency is best implemented when lock times will be short, as in programmatic processing of records. Pessimistic concurrency is not a scalable option when users are interacting with data, causing records to be locked for relatively large periods of time.

Pessimistic concurrency control is not possible in a disconnected architecture. Connections are open only long enough to read the data or to update it, so locks cannot be sustained for long periods. Moreover, an application that holds onto locks for long periods is not scalable.

Optimistic Concurrency

In optimistic concurrency, locks are set and held only while the database is being accessed. The locks prevent other users from attempting to update records at the same instant. The data is always available except for the exact moment that an update is taking place.

When an update is attempted, the original version of a changed row is compared against the existing row in the database. If the two are different, the update fails with a concurrency error. It is up to you at that point to reconcile the two rows, using business logic that you create.

Users who use optimistic concurrency do not lock a row when reading it. When a user wants to update a row, the application must determine whether another user has changed the row since it was read. Optimistic concurrency is generally used in environments with a low contention for data. This improves performance as no locking of records is required, and locking of records requires additional server resources. Also, in order to maintain record locks, a persistent connection to the database server is required. Because this is not the case in an optimistic concurrency model, connections to the server are free to serve a larger number of clients in less time.

In an optimistic concurrency model, a violation is considered to have occurred if, after a user receives a value from the database, another user modifies the value before the first user has attempted to modify it.

The following tables follow an example of optimistic concurrency.

At 1:00 p.m., User1 reads a row from the database with the following values:

CustID	***LastName***	***FirstName***
101	Smith	Bob

Column name	***Original value***	***Current value***	***Value in database***
CustID	101	101	101
LastName	Smith	Smith	Smith
FirstName	Bob	Bob	Bob

At 1:01 p.m., User2 reads the same row.

At 1:03 p.m., User2 changes **FirstName** from "Bob" to "Robert" and updates the database.

Column name	***Original value***	***Current value***	***Value in database***
CustID	101	101	101
LastName	Smith	Smith	Smith
FirstName	Bob	Robert	Bob

The update succeeds because the values in the database at the time of update match the original values that User2 has.

At 1:05 p.m., User1 changes Bob's first name to "James" and tries to update the row.

Column name	*Original value*	*Current value*	*Value in database*
CustID	101	101	101
LastName	Smith	Smith	Smith
FirstName	Bob	James	Robert

At this point, User1 encounters an optimistic concurrency violation because the values in the database no longer match the original values that User1 was expecting. The decision now needs to be made whether to overwrite the changes supplied by User2 with the changes supplied by User1 or to cancel the changes by User1.

Testing for Optimistic Concurrency Violations

There are several techniques for testing for an optimistic concurrency violation. One involves including a ***timestamp*** column in the table. Databases commonly provide timestamp functionality that can be used to identify the date and time when the record was last updated. Using this technique, a timestamp column is included in the table definition. Whenever the record is updated, the timestamp is updated to reflect the current date and time. In a test for optimistic concurrency violations, the timestamp column is returned with any query of the contents of the table. When an update is attempted, the timestamp value in the database is compared to the original timestamp value contained in the modified row. If they match, the update is performed and the timestamp column is updated with the current time to reflect the update. If they do not match, an optimistic concurrency violation has occurred.

Consider following example.

Supposed Juliet and Romeo are working with the same data. If Juliet updates a row after Romeo reads it but before he posts his own changes, Romeo's update attempt will fail because it will use the original timestamp value to try to locate the row. Romeo's *UPDATE* statement will include the timestamp column in its *WHERE* clause but won't be able to locate the original record because the timestamp value has changed due to Juliet's update. This prevents Romeo from overwriting Juliet's changes and provides a means for his application to detect that another user modified the row he was editing.

The *TSEQUAL()* function can be used to compare values. If the timestamps aren't equal, *TSEQUAL()* raises an error and aborts the current command batch.

Table is limited to a single timestamp column. A common convention is to name the column ***timestamp***, but that's not required by the server. Here's a code sample that shows how to use timestamp data type:

```
SET NONCOUNT ON
CREATE TABLE #testts (c1 int identity, c2 int DEFAULT 0, changelog timestamp)

INSERT #testts DEFAULT VALUES
INSERT #testts DEFAULT VALUES
INSERT #testts DEFAULT VALUES
INSERT #testts DEFAULT VALUES
INSERT #testts DEFAULT VALUES

SELECT * FROM #testts

UPDATE #testts SET c2=c1
```

```
SELECT * FROM #testts

GO
DROP TABLE #testts
```

c1	c2	changelog
1	0	0x0000000000000085
2	0	0x0000000000000086
3	0	0x0000000000000087
4	0	0x0000000000000088
5	0	0x0000000000000089

c1	c2	changelog
1	1	0x000000000000008A
2	2	0x000000000000008B
3	3	0x000000000000008C
4	4	0x000000000000008D
5	5	0x000000000000008E

Note the different values for each row's timestamp column before and after the *UPDATE*.

You can access the last generated timestamp value for a database via the *@@DBTS* automatic variable. Each database maintains its own counter, so be sure you're in the correct database before querying *@@DBTS*. Here's an example:

```
USE tempdb
GO
SET NONCOUNT ON
CREATE TABLE #testts (c1 int identity, c2 int DEFAULT 0, changelog timestamp)

INSERT #testts DEFAULT VALUES
INSERT #testts DEFAULT VALUES
INSERT #testts DEFAULT VALUES
INSERT #testts DEFAULT VALUES
INSERT #testts DEFAULT VALUES

SELECT * FROM #testts
SELECT @@DBTS AS 'Last timestamp'

UPDATE #testts SET c2=c1

SELECT * FROM #testts
SELECT @@DBTS AS 'Last timestamp'

GO
```

DROP TABLE #testts

c1	*c2*	*changelog*
1	0	0x00000000000000B7
2	0	0x00000000000000B8
3	0	0x00000000000000B9
4	0	0x00000000000000BA
5	0	0x00000000000000BB

Last timestamp

0x00000000000000BB

c1	*c2*	*changelog*
1	1	0x00000000000000BC
2	2	0x00000000000000BD
3	3	0x00000000000000BE
4	4	0x00000000000000BF
5	5	0x00000000000000C0

Last timestamp

0x00000000000000C0

Note the *USE* tempdb at the first of the script. Since temporary tables reside in tempdb, we have to change the current database focus to tempdb in order for *@@DBTS* to work properly. *@@DBTS* always returns the last timestamp value generated for a database, so you can use it to acquire the timestamp of an update you've just performed.

Another technique for testing for an optimistic concurrency violation is to verify that all the original column values in a row still match those found in the database. For example, consider the following query:

SELECT Col1, Col2, Col3 FROM Table1

To test for an optimistic concurrency violation when updating a row in *Table1*, you would issue the following *UPDATE* statement:

```
UPDATE Table1 Set Col1 = @NewCol1Value,
              Set Col2 = @NewCol2Value,
              Set Col3 = @NewCol3Value
WHERE Col1 = @OldCol1Value AND
    Col2 = @OldCol2Value AND
    Col3 = @OldCol3Value
```

As long as the original values match the values in the database, the update is performed. If a value has been modified, the update will not modify the row because the *WHERE* clause will not find a match.

Note that it is recommended to always return a unique primary key value in your query. Otherwise, the preceding *UPDATE* statement may update more than one row, which might not be your intent.

If a column at your data source allows nulls, you may need to extend your *WHERE* clause to check for a matching null reference in your local table and at the data source. For example, the following UPDATE statement verifies that a null reference in the local row still matches a null reference at the data source, or that the value in the local row still matches the value at the data source.

```
UPDATE Table1 Set Col1 = @NewVal1
WHERE (@OldVal1 IS NULL AND Col1 IS NULL) OR Col1 = @OldVal1
```

You may also choose to apply less restrictive criteria when using an optimistic concurrency model. For example, using only the primary key columns in the *WHERE* clause results in the data being overwritten regardless of whether the other columns have been updated since the last query. You can also apply a *WHERE* clause only to specific columns, resulting in data being overwritten unless particular fields have been updated since they were last queried.

Optimistic Concurrency Example

The following is a simple example that sets the *UpdateCommand* of a *DataAdapter* to test for optimistic concurrency, and then uses the *RowUpdated* event to test for optimistic concurrency violations. When an optimistic concurrency violation is encountered, the application sets the *RowError* of the row that the update was issued for to reflect an optimistic concurrency violation.

Note that the parameter values passed to the *WHERE* clause of the *UPDATE* command are mapped to the *Original* values of their respective columns.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Web.Configuration;

public partial class _Default : System.Web.UI.Page
{
   SqlConnection conn;
   SqlCommand cmd;
   SqlDataReader reader;
```

```
SqlDataAdapter custDA;

protected void Page_Load(object sender, EventArgs e)
{
   conn = new SqlConnection(WebConfigurationManager.
       ConnectionStrings["ConnectionString"].ConnectionString);

   //conn = new SqlConnection("Data Source=localhost;Integrated Security=SSPI;Initial
       Catalog=Halloween");

   custDA = new SqlDataAdapter(
      "SELECT ProductID, Name FROM Products ORDER BY ProductID", conn);

   // The Update command checks for optimistic concurrency violations in the WHERE clause.
   custDA.UpdateCommand = new SqlCommand(
      "UPDATE Products (ProductID,  Name) " +
                "VALUES(@ProductID, @Name) " +
      "WHERE ProductID = @oldProductID AND Name = @oldName", conn);

   custDA.UpdateCommand.Parameters.Add("@ProductID", SqlDbType.Char, 10, "ProductID");
   custDA.UpdateCommand.Parameters.Add("@Name", SqlDbType.VarChar, 50, "Name");

   // Pass the original values to the WHERE clause parameters.
   SqlParameter myParm;

   myParm = custDA.UpdateCommand.Parameters.Add(
      "@oldProductID", SqlDbType.Char, 10, "ProductID");
   myParm.SourceVersion = DataRowVersion.Original;

   myParm = custDA.UpdateCommand.Parameters.Add(
      "@oldName", SqlDbType.VarChar, 50, "Name");
   myParm.SourceVersion = DataRowVersion.Original;

   // Add the RowUpdated event handler.
   custDA.RowUpdated += new SqlRowUpdatedEventHandler(OnRowUpdated);

   DataSet custDS = new DataSet();

   custDA.Fill(custDS, "Products");

   // Modify the DataSet contents.
   custDA.Update(custDS, "Products");

   foreach (DataRow myRow in custDS.Tables["Products"].Rows)
   {
       if (myRow.HasErrors)
```

```
            Console.WriteLine(myRow[0] + "\n" + myRow.RowError);
        }
    }

    protected static void OnRowUpdated(object sender, SqlRowUpdatedEventArgs args)
    {
        if (args.RecordsAffected == 0)
        {
            args.Row.RowError = "Optimistic Concurrency Violation _ Encountered";
            args.Status = UpdateStatus.SkipCurrentRow;
        }
    }
}
```

Last in Wins

With "*last in wins*," no check of the original data is made and the update is simply written to the database. It is understood that the following scenario can occur:

- User A fetches a record from the database.
- User B fetches the same record from the database, modifies it, and writes the updated record back to the database.
- User A modifies the 'old' record and writes it back to the database.

In the above scenario, the changes User B made were never seen by User A. Be sure that this situation is acceptable if you plan to use the ***"last in wins"*** approach of concurrency control.

ADO.NET classes

The *SqlConnection* class

Before you can access the data in a database, you have to create a connection object that defines the connection to the database. To do that, you use the *SqlConnection* class presented below

Property	**Description**
ConnectionString	Contains information that lets you connect to a SQL Server database. The connection string includes information such as name of the server, the name of the database, and login information
Method	**Description**
Open	Opens a connection to a database
Close	Closes a connection to a database

The *SqlCommand* class

To execute a SQL statement against a SQL Server database, you create a *SqlCommand* object that contains the statement. Table below presents the *SqlCommand* class you use to create this object

Property	Description
Connection	The *SqlConnection* object that's used by the command to connect to the database.
CommandText	The text of the SQL command or the name of a stored procedure or database table
CommandType	A constant in the *CommandType* enumeration that indicates whether the *CommandText* property contains a SQL statement *(Text),* the name of a stored procedures *(StoredProcedure),* or the name of a database table *(TableDirect)*
Parameters	The collection of parameters used by the command

Method	Description
ExecuteReader	Executes a query and returns the result as a *SqlDataReader* object
ExecuteNonquery	Executes the command and returns an integer representing the number of rows affected. This method you use for executing command *Insert, Update*, or *Delete.*
ExecuteScalar	Executes a query and returns the first column of the first row returned by the query

Common properties of the *SqlParameter* class

Property	Description
ParameterName	The name of the parameter.
Value	The value assigned to the parameter.
SqlDbType	The SQL data type for the parameter.

The *SqlDataReader* class

A data reader provides an efficient way of reading the rows in a result set returned by a database query. A data reader lets you read rows but not modify them. In other words, a ***data reader is read-only and lets you read rows in a forward direction only.*** Once you read the next row, the previous row is unavailable. Below is the list of the most important properties and methods of the *SqlDataReader* class.

Property	Description
Item	Access the column with the specified index or name from the current row
FieldCount	The number of columns in the current row

Method	Description
Read	Reads the next row. Returns *True* if there are more rows. Otherwise, returns *False.*
Close	Closes the data reader

The *SqlDataAdapter* class

The job of a data adapter is to provide a link between a dataset and a database. Below are the properties of the *SqlDataAdapter* class identify the four SQL commands that the data adapter uses to transfer data from the database to the dataset and vice versa.

Property	Description
SelectCommand	A *SqlCommand* object representing the *Select* statement used to query the database
DeleteCommand	A *SqlCommand* object representing the *Delete* statement used to delete a row from the database
InsertCommand	A *SqlCommand* object representing the *Insert* statement used to add a row to the database
UpdateCommand	A *SqlCommand* object representing the *Update* statement used to update a row in the database

Method	Description
Fill	Executes the command identified by the *SelectCommand* property and loads the result into a dataset object
Update	Executes the command identified by the *DeleteCommand*, *InsertCommand*, and *UpdateCommand* properties for each row in the dataset that was deleted, added, or updated