Handling Data Concurrency Using ADO.NET

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifContents

[Is Your Glass Half Full?](http://msdn.microsoft.com/en-us/magazine/cc163924.aspx#S1)  
[Where Did My Changes Go?](http://msdn.microsoft.com/en-us/magazine/cc163924.aspx#S2)  
[Detecting Violations](http://msdn.microsoft.com/en-us/magazine/cc163924.aspx#S3)  
[Retrieving Row Flags](http://msdn.microsoft.com/en-us/magazine/cc163924.aspx#S4)  
[Saving Changes](http://msdn.microsoft.com/en-us/magazine/cc163924.aspx#S5)  
[Reporting Violations](http://msdn.microsoft.com/en-us/magazine/cc163924.aspx#S6)  
[User's Choice](http://msdn.microsoft.com/en-us/magazine/cc163924.aspx#S7)  
[Wrapping It Up](http://msdn.microsoft.com/en-us/magazine/cc163924.aspx#S8)

One of the key features of the ADO.NET DataSet is that it can be a self-contained and disconnected data store. It can contain the schema and data from several rowsets in DataTable objects as well as information about how to relate the DataTable objects—all in memory. The DataSet neither knows nor cares where the data came from, nor does it need a link to an underlying data source. Because it is data source agnostic you can pass the DataSet around networks or even serialize it to XML and pass it across the Internet without losing any of its features. However, in a disconnected model, concurrency obviously becomes a much bigger problem than it is in a connected model.

In this column, I'll explore how ADO.NET is equipped to detect and handle concurrency violations. I'll begin by discussing scenarios in which concurrency violations can occur using the ADO.NET disconnected model. Then I will walk through an ASP.NET application that handles concurrency violations by giving the user the choice to overwrite the changes or to refresh the out-of-sync data and begin editing again. Because part of managing an optimistic concurrency model can involve keeping a timestamp (rowversion) or another type of flag that indicates when a row was last updated, I will show how to implement this type of flag and how to maintain its value after each database update.

**Is Your Glass Half Full?**

There are three common techniques for managing what happens when users try to modify the same data at the same time: pessimistic, optimistic, and last-in wins. They each handle concurrency issues differently.

The pessimistic approach says: "Nobody can cause a concurrency violation with my data if I do not let them get at the data while I have it." This tactic prevents concurrency in the first place but it limits scalability because it prevents all concurrent access. Pessimistic concurrency generally locks a row from the time it is retrieved until the time updates are flushed to the database. Since this requires a connection to remain open during the entire process, pessimistic concurrency cannot successfully be implemented in a disconnected model like the ADO.NET DataSet, which opens a connection only long enough to populate the DataSet then releases and closes, so a database lock cannot be held.

Another technique for dealing with concurrency is the last-in wins approach. This model is pretty straightforward and easy to implement—whatever data modification was made last is what gets written to the database. To implement this technique you only need to put the primary key fields of the row in the UPDATE statement's WHERE clause. No matter what is changed, the UPDATE statement will overwrite the changes with its own changes since all it is looking for is the row that matches the primary key values. Unlike the pessimistic model, the last-in wins approach allows users to read the data while it is being edited on screen. However, problems can occur when users try to modify the same data at the same time because users can overwrite each other's changes without being notified of the collision. The last-in wins approach does not detect or notify the user of violations because it does not care. However the optimistic technique does detect violations.

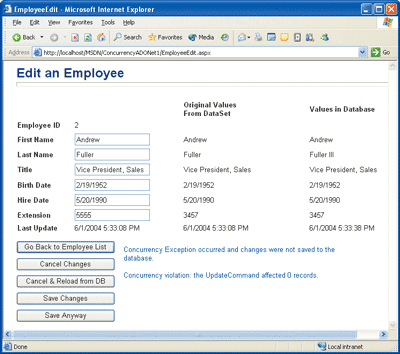


Figure 1 **Concurrency Violation**

In optimistic concurrency models, a row is only locked during the update to the database. Therefore the data can be retrieved and updated by other users at any time other than during the actual row update operation. Optimistic concurrency allows the data to be read simultaneously by multiple users and blocks other users less often than its pessimistic counterpart, making it a good choice for ADO.NET. In optimistic models, it is important to implement some type of concurrency violation detection that will catch any additional attempt to modify records that have already been modified but not committed. You can write your code to handle the violation by always rejecting and canceling the change request or by overwriting the request based on some business rules. Another way to handle the concurrency violation is to let the user decide what to do. The sample application that is shown in **Figure 1** illustrates some of the options that can be presented to the user in the event of a concurrency violation.

**Where Did My Changes Go?**

When users are likely to overwrite each other's changes, control mechanisms should be put in place. Otherwise, changes could be lost. If the technique you're using is the last-in wins approach, then these types of overwrites are entirely possible.

For example, imagine Julie wants to edit an employee's last name to correct the spelling. She navigates to a screen which loads the employee's information into a DataSet and has it presented to her in a Web page. Meanwhile, Scott is notified that the same employee's phone extension has changed. While Julie is correcting the employee's last name, Scott begins to correct his extension. Julie saves her changes first and then Scott saves his.

Assuming that the application uses the last-in wins approach and updates the row using a SQL WHERE clause containing only the primary key's value, and assuming a change to one column requires the entire row to be updated, neither Julie nor Scott may immediatelyrealize the concurrency issue that just occurred. In this particular situation, Julie's changes were overwritten by Scott's changes because he saved last, and the last name reverted to the misspelled version.

So as you can see, even though the users changed different fields, their changes collided and caused Julie's changes to be lost. Without some sort of concurrency detection and handling, these types of overwrites can occur and even go unnoticed.

When you run the sample application included in this column's download, you should open two separate instances of Microsoft® Internet Explorer. When I generated the conflict, I opened two instances to simulate two users with two separate sessions so that a concurrency violation would occur in the sample application. When you do this, be careful not to use Ctrl+N because if you open one instance and then use the Ctrl+N technique to open another instance, both windows will share the same session.

**Detecting Violations**

The concurrency violation reported to the user in **Figure 1** demonstrates what can happen when multiple users edit the same data at the same time. In **Figure 1**, the user attempted to modify the first name to "Joe" but since someone else had already modified the last name to "Fuller III," a concurrency violation was detected and reported. ADO.NET detects a concurrency violation when a DataSet containing changed values is passed to a SqlDataAdapter's Update method and no rows are actually modified. Simply using the primary key (in this case the EmployeeID) in the UPDATE statement's WHERE clause will not cause a violation to be detected because it still updates the row (in fact, this technique has the same outcome as the last-in wins technique). Instead, more conditions must be specified in the WHERE clause in order for ADO.NET to detect the violation.

The key here is to make the WHERE clause explicit enough so that it not only checks the primary key but that it also checks for another appropriate condition. One way to accomplish this is to pass in all modifiable fields to the WHERE clause in addition to the primary key. For example, the application shown in **Figure 1** could have its UPDATE statement look like the stored procedure that's shown in **Figure 2**.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 2 Putting All Fields in the WHERE Clause

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl14_ctl00_ctl00_code');" \o "Copy Code)

CREATE PROCEDURE prUpd\_EmployeeTest1

@LastName nvarchar(20), @FirstName nvarchar(10),

@Title nvarchar(30), @BirthDate datetime,

@HireDate datetime, @Extension nvarchar(4),

@Original\_EmployeeID int, @Original\_BirthDate datetime,

@Original\_Extension nvarchar(4), @Original\_FirstName nvarchar(10),

@Original\_HireDate datetime, @Original\_LastName nvarchar(20),

@Original\_Title nvarchar(30), @EmployeeID int

AS

UPDATE Employees

SET LastName = @LastName, FirstName = @FirstName,

Title = @Title, BirthDate = @BirthDate,

HireDate = @HireDate, Extension = @Extension

WHERE (EmployeeID = @Original\_EmployeeID)

AND (FirstName = @Original\_FirstName)

AND (LastName = @Original\_LastName)

AND (BirthDate = @Original\_BirthDate OR @Original\_BirthDate IS NULL

AND BirthDate IS NULL)

AND (Extension = @Original\_Extension OR @Original\_Extension IS NULL

AND Extension IS NULL)

AND (HireDate = @Original\_HireDate OR @Original\_HireDate IS NULL

AND HireDate IS NULL)

AND (Title = @Original\_Title OR @Original\_Title IS NULL AND Title

IS NULL);

Notice that in the code in **Figure 2** nullable columns are also checked to see if the value passed in is NULL. This technique is not only messy but it can be difficult to maintain by hand and it requires you to test for a significant number of WHERE conditions just to update a row. This yields the desired result of only updating rows where none of the values have changed since the last time the user got the data, but there are other techniques that do not require such a huge WHERE clause.

Another way to make sure that the row is only updated if it has not been modified by another user since you got the data is to add a timestamp column to the table. The SQL Server™ TIMESTAMP datatype automatically updates itself with a new value every time a value in its row is modified. This makes it a very simple and convenient tool to help detect concurrency violations.

A third technique is to use a DATETIME column in which to track changes to its row. In my sample application I added a column called LastUpdateDateTime to the Employees table.

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl06_code');" \o "Copy Code)

ALTER TABLE Employees ADD LastUpdateDateTime DATETIME

There I update the value of the LastUpdateDateTime field automatically in the UPDATE stored procedure using the built-in SQL Server GETDATE function.

The binary TIMESTAMP column is simple to create and use since it automatically regenerates its value each time its row is modified, but since the DATETIME column technique is easier to display on screen and demonstrate when the change was made, I chose it for my sample application. Both of these are solid choices, but I prefer the TIMESTAMP technique since it does not involve any additional code to update its value.

**Retrieving Row Flags**

One of the keys to implementing concurrency controls is to update the timestamp or datetime field's value back into the DataSet. If the same user wants to make more modifications, this updated value is reflected in the DataSet so it can be used again. There are a few different ways to do this. The fastest is using output parameters within the stored procedure. (This should only return if @@ROWCOUNT equals 1.) The next fastest involves selecting the row again after the UPDATE within the stored procedure. The slowest involves selecting the row from another SQL statement or stored procedure from the SqlDataAdapter's RowUpdated event.

I prefer to use the output parameter technique since it is the fastest and incurs the least overhead. Using the RowUpdated event works well, but it requires me to make a second call from the application to the database. The following code snippet adds an output parameter to the SqlCommand object that is used to update the Employee information:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl07_code');" \o "Copy Code)

oUpdCmd.Parameters.Add(new SqlParameter("@NewLastUpdateDateTime",

SqlDbType.DateTime, 8, ParameterDirection.Output,

false, 0, 0, "LastUpdateDateTime", DataRowVersion.Current, null));

oUpdCmd.UpdatedRowSource = UpdateRowSource.OutputParameters;

The output parameter has its sourcecolumn and sourceversion arguments set to point the output parameter's return value back to the current value of the LastUpdateDateTime column of the DataSet. This way the updated DATETIME value is retrieved and can be returned to the user's .aspx page.

**Saving Changes**

Now that the Employees table has the tracking field (LastUpdateDateTime) and the stored procedure has been created to use both the primary key and the tracking field in the WHERE clause of the UPDATE statement, let's take a look at the role of ADO.NET. In order to trap the event when the user changes the values in the textboxes, I created an event handler for the TextChanged event for each TextBox control:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl08_code');" \o "Copy Code)

private void txtLastName\_TextChanged(object sender, System.EventArgs e)

{

// Get the employee DataRow (there is only 1 row, otherwise I could

// do a Find)

dsEmployee.EmployeeRow oEmpRow =

(dsEmployee.EmployeeRow)oDsEmployee.Employee.Rows[0];

oEmpRow.LastName = txtLastName.Text;

// Save changes back to Session

Session["oDsEmployee"] = oDsEmployee;

}

This event retrieves the row and sets the appropriate field's value from the TextBox. (Another way of getting the changed values is to grab them when the user clicks the Save button.) Each TextChanged event executes after the Page\_Load event fires on a postback, so assuming the user changed the first and last names, when the user clicks the Save button, the events could fire in this order: Page\_Load, txtFirstName\_TextChanged, txtLastName\_TextChanged, and btnSave\_Click.

The Page\_Load event grabs the row from the DataSet in the Session object; the TextChanged events update the DataRow with the new values; and the btnSave\_Click event attempts to save the record to the database. The btnSave\_Click event calls the SaveEmployee method (shown in **Figure 3**) and passes it a bLastInWins value of false since we want to attempt a standard save first. If the SaveEmployee method detects that changes were made to the row (using the HasChanges method on the DataSet, or alternatively using the RowState property on the row), it creates an instance of the Employee class and passes the DataSet to its SaveEmployee method. The Employee class could live in a logical or physical middle tier. (I wanted to make this a separate class so it would be easy to pull the code out and separate it from the presentation logic.)

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 3 Saving the Changes

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl18_ctl00_ctl00_code');" \o "Copy Code)

private void SaveEmployee(bool bLastInWins)

{

try

{

// If changes were made, then attempt to save the record

if (oDsEmployee.HasChanges(DataRowState.Modified))

{

// Create an instance of the Employee object

Employee oEmp = new Employee();

// Save the changes to the database

dsEmployee oDsAfterUpdate = oEmp.SaveEmployee(oDsEmployee,

bLastInWins);

// Merge the new LastUpdateDateTime field value back into the

// local DataSet

oDsEmployee.Merge(oDsAfterUpdate);

// Display success message

lblMessage.Text = "Changes have been applied successfully";

}

else

{

// Display "nothing to do" message

lblMessage.Text = "No modifications were made," +

"nothing to save";

}

}

catch (DBConcurrencyException exDBC)

{

// In addition to displaying the current values in the DataSet,

// display the original values from the DataSet from when it was

// first loaded and the values that are currently in the

// database.

FillConcurrencyValues((dsEmployee.EmployeeRow)exDBC.Row);

btnSaveAnyway.Visible = true;

// Display error message

string sMsg = "Concurrency Exception occurred and changes" +

"were not saved to the database." +

"<br><br>" + exDBC.Message;

lblMessage.Text = sMsg;

}

catch (Exception ex)

{

// Display error message

lblMessage.Text = "Exception occurred and changes were" +

"not saved to the database." +

"<br><br>" + ex.Message;

}

finally

{

// Bind the DataSet to the controls on the form

BindControls();

}

}

Notice that I did not use the GetChanges method to pull out only the modified rows and pass them to the Employee object's Save method. I skipped this step here since there is only one row. However, if there were multiple rows in the DataSet's DataTable, it would be better to use the GetChanges method to create a DataSet that contains only the modified rows.

If the save succeeds, the Employee.SaveEmployee method returns a DataSet containing the modified row and its newly updated row version flag (in this case, the LastUpdateDateTime field's value). This DataSet is then merged into the original DataSet so that the LastUpdateDateTime field's value can be updated in the original DataSet. This must be done because if the user wants to make more changes she will need the current values from the database merged back into the local DataSet and shown on screen. This includes the LastUpdateDateTime value which is used in the WHERE clause. Without this field's current value, a false concurrency violation would occur.

**Reporting Violations**

If a concurrency violation occurs, it will bubble up and be caught by the exception handler shown in **Figure 3** in the catch block for DBConcurrencyException. This block calls the FillConcurrencyValues method, which displays both the original values in the DataSet that were attempted to be saved to the database and the values currently in the database. This method is used merely to show the user why the violation occurred. Notice that the exDBC variable is passed to the FillConcurrencyValues method. This instance of the special database concurrency exception class (DBConcurrencyException) contains the row where the violation occurred. When a concurrency violation occurs, the screen is updated to look like **Figure 1**.

The DataSet not only stores the schema and the current data, it also tracks changes that have been made to its data. It knows which rows have been modified and it keeps track of the before and after versions of these values. When accessing a column's value via the DataRow's indexer, in addition to the column index you can also specify a value using the DataRowVersion enumerator. For example, after a user changes the value of the last name of an employee, the following lines of C# code will retrieve the original and current values stored in the LastName column:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl11_code');" \o "Copy Code)

string sLastName\_Before = oEmpRow["LastName", DataRowVersion.Original];

string sLastName\_After = oEmpRow["LastName", DataRowVersion.Current];

The FillConcurrencyValues method uses the row from the DBConcurrencyException and gets a fresh copy of the same row from the database. It then displays the values using the DataRowVersion enumerators to show the original value of the row before the update and the value in the database alongside the current values in the textboxes.

**User's Choice**

Once the user has been notified of the concurrency issue, you could leave it up to her to decide how to handle it. Another alternative is to code a specific way to deal with concurrency, such as always handling the exception to let the user know (but refreshing the data from the database). In this sample application I let the user decide what to do next. She can either cancel changes, cancel and reload from the database, save changes, or save anyway.

The option to cancel changes simply calls the RejectChanges method of the DataSet and rebinds the DataSet to the controls in the ASP.NET page. The RejectChanges method reverts the changes that the user made back to its original state by setting all of the current field values to the original field values. The option to cancel changes and reload the data from the database also rejects the changes but additionally goes back to the database via the Employee class in order to get a fresh copy of the data before rebinding to the control on the ASP.NET page.

The option to save changes attempts to save the changes but will fail if a concurrency violation is encountered. Finally, I included a "save anyway" option. This option takes the values the user attempted to save and uses the last-in wins technique, overwriting whatever is in the database. It does this by calling a different command object associated with a stored procedure that only uses the primary key field (EmployeeID) in the WHERE clause of the UPDATE statement. This technique should be used with caution as it will overwrite the record.

If you want a more automatic way of dealing with the changes, you could get a fresh copy from the database. Then overwrite just the fields that the current user modified, such as the Extension field. That way, in the example I used the proper LastName would not be overwritten. Use this with caution as well, however, because if the same field was modified by both users, you may want to just back out or ask the user what to do next. What is obvious here is that there are several ways to deal with concurrency violations, each of which must be carefully weighed before you decide on the one you will use in your application.

**Wrapping It Up**

Setting the SqlDataAdapter's ContinueUpdateOnError property tells the SqlDataAdapter to either throw an exception when a concurrency violation occurs or to skip the row that caused the violation and to continue with the remaining updates. By setting this property to false (its default value), it will throw an exception when it encounters a concurrency violation. This technique is ideal when only saving a single row or when you are attempting to save multiple rows and want them all to commit or all to fail.

I have split the topic of concurrency violation management into two parts. Next time I will focus on what to do when multiple rows could cause concurrency violations. I will also discuss how the DataViewRowState enumerators can be used to show what changes have been made to a DataSet.