**Transactions and Concurrency (ADO.NET)**

This page is specific to

**Microsoft Visual Studio 2008/.NET Framework 3.5**

Other versions are also available for the following:

[Microsoft Visual Studio 2003/.NET Framework 1.1](http://msdn.microsoft.com/en-us/library/777e5ebh(VS.71).aspx)

[Microsoft Visual Studio 2005/.NET Framework 2.0](http://msdn.microsoft.com/en-us/library/777e5ebh(VS.80).aspx)

[.NET Framework 3.0](http://msdn.microsoft.com/en-us/library/777e5ebh(VS.85).aspx)

[Microsoft Visual Studio 2010/.NET Framework 4.0](http://msdn.microsoft.com/en-us/library/777e5ebh(VS.100).aspx)

A transaction consists of a single command or a group of commands that execute as a package. Transactions allow you to combine multiple operations into a single unit of work. If a failure occurs at one point in the transaction, all of the updates can be rolled back to their pre-transaction state.

A transaction must conform to the ACID properties—atomicity, consistency, isolation, and durability—in order to guarantee data consistency. Most relational database systems, such as Microsoft SQL Server, support transactions by providing locking, logging, and transaction management facilities whenever a client application performs an update, insert, or delete operation.

|  |
| --- |
| **NoteNote:** |
| Transactions that involve multiple resources can lower concurrency if locks are held too long. Therefore, keep transactions as short as possible. |

If a transaction involves multiple tables in the same database or server, then explicit transactions in stored procedures often perform better. You can create transactions in SQL Server stored procedures by using the Transact-SQL **BEGIN TRANSACTION**, **COMMIT TRANSACTION**, and **ROLLBACK TRANSACTION** statements. For more information, see SQL Server Books Online.

Transactions involving different resource managers, such as a transaction between SQL Server and Oracle, require a distributed transaction.

http://i.msdn.microsoft.com/Global/Images/clear.gif In This Section

[Local Transactions (ADO.NET)](http://msdn.microsoft.com/en-us/library/2k2hy99x.aspx)

Demonstrates how to perform transactions against a database.

[Distributed Transactions (ADO.NET)](http://msdn.microsoft.com/en-us/library/ms254973.aspx)

Describes how to perform distributed transactions in ADO.NET.

[System.Transactions Integration with SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/ms172070.aspx)

Describes [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) integration with SQL Server 2005 for working with distributed transactions.

[Optimistic Concurrency (ADO.NET)](http://msdn.microsoft.com/en-us/library/aa0416cz.aspx)

Describes optimistic and pessimistic concurrency, and how you can test for concurrency violations.

**Transaction Fundamentals**

Transactions bind multiple tasks together. For example, imagine that an application performs two tasks. First, it creates a new table in a database. Next, it calls a specialized object to collect, format, and insert data into the new table. These two tasks are related and even interdependent, such that you want to avoid creating a new table unless you can fill it with data. Executing both tasks within the scope of a single transaction enforces the connection between them. If the second task fails, the first task is rolled back to a point before the new table was created.

To ensure predictable behavior, all transactions must possess the basic ACID properties (atomic, consistent, isolated, and durable). These properties reinforce the role of mission-critical transactions as all-or-none propositions. For more information on ACID, please see [ACID Properties](http://go.microsoft.com/fwlink/?LinkId=98791). In summary, ACID guarantees that a set of related tasks either succeed or fail as a unit. In transaction processing terminology, the transaction either commits or aborts. For a transaction to commit, all participants must guarantee that any change to data will be permanent. Changes must persist despite system crashes or other unforeseen events. If even a single participant fails to make this guarantee, the entire transaction fails. All changes to data within the scope of the transaction are rolled back to a specific set point.

A transaction can be confined to a single data resource, such as a database or message queue. In this scenario, the local transaction is managed by the Transaction Manager provided by [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) , which generates performance gain. Controlled by the data resource, these transactions are efficient and easy to manage.

Transactions can also span multiple data resources. Distributed transactions give you the ability to incorporate several distinct operations occurring on different systems into a single pass or fail action. In this scenario, the transactions are coordinated by the Microsoft Distributed Transaction Coordinator (MSDTC) that resides in each system.

When you develop a transactional application using the classes provided by **System.Transactions**, you do not need to worry about what kind of transactions you need, or the transaction manager involved. The **System.Transactions** infrastructure automatically manages these for you.

When you create a transaction, you can specify the isolation level that applies to the transaction. The isolation level, defined by the [IsolationLevel](http://msdn.microsoft.com/en-us/library/system.transactions.isolationlevel.aspx) class, determines what level of access other transactions will have to the data affected by your transaction.

You can create transactions using ADO.NET, [System.EnterpriseServices](http://msdn.microsoft.com/en-us/library/system.enterpriseservices.aspx), or the new transactional programming model provided by the **System.Transactions** namespace. The [Features Provided by System.Transactions](http://msdn.microsoft.com/en-us/library/0abf6ykb.aspx) section discusses the features that you can use to write a transactional application using the **System.Transactions** namespace.

**Features Provided by System.Transactions**

This section describes how you can use the features provided by the [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) namespace to write your own transactional application and resource manager. Specifically, this section covers how to create and participate in a transaction (local or distributed) with one or multiple participants.

## Overview of System.Transactions

The infrastructure provided by the classes in the **System.Transactions** namespace makes transactional programming simple and efficient by supporting transactions initiated in SQL Server, ADO.NET, Message Queuing (MSMQ), and the Microsoft Distributed Transaction Coordinator (MSDTC). The **System.Transactions** namespace provides both an explicit programming model based on the [Transaction](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.aspx) class, as well as an implicit programming model using the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) class, in which transactions are automatically managed by the infrastructure. For more information on how to create a transactional application using these two models, see [Writing a Transactional Application](http://msdn.microsoft.com/en-us/library/ms229973.aspx).

The **System.Transactions** namespace also provides types for you to implement a resource manager. A resource manager manages durable or volatile data used in a transaction, and work in cooperation with the transaction manager to provide the application with a guarantee of atomicity and isolation. The transaction manager that is provided by the **System.Transactions** infrastructure supports transactions involving multiple volatile resources or a single durable resource. For more information on implementing a resource manager, see [Implementing a Resource Manager](http://msdn.microsoft.com/en-us/library/ms229975.aspx).

The transaction manager also transparently escalates local transactions to distributed transactions by coordinating with a disk-based transaction manager like the DTC, when an additional durable resource manager enlists itself with a transaction. There are two key ways that the **System.Transactions** infrastructure provides enhanced performance.

* Dynamic Escalation, which ensures that the **System.Transactions** infrastructure only engages the MSDTC when a transaction spans across multiple distributed resources. For more information about dynamic escalation. see [Transaction Management Escalation](http://msdn.microsoft.com/en-us/library/ms229978.aspx) topic.
* Promotable Enlistments, which allows a resource, such as a database, to take ownership of the transaction if it is the only entity participating in the transaction. Later, if needed, the **System.Transactions** infrastructure can still escalate the management of the transaction to the MSDTC. This further reduces the chance of using the MSDTC. Promotable Enlistments are covered in depth in the topic[Optimization using Single Phase Commit and Promotable Single Phase Notification](http://msdn.microsoft.com/en-us/library/ms229980.aspx).

The **System.Transactions** namespace defines three levels of trust - AllowPartiallyTrustedCallers (APTCA), DistributedTransactionPermission(DTP) and full trust - that restrict access to the types of resources it exposes. For more information on the various trust levels, see [Security Trust Levels in Accessing Resources](http://msdn.microsoft.com/en-us/library/ms229983.aspx).

## In this section

### Writing A Transactional Application

The **System.Transactions** namespace provides two models for creating transactional applications. [Implementing an Implicit Transaction using Transaction Scope](http://msdn.microsoft.com/en-us/library/ms172152.aspx) describes how the **System.Transactions** namespace supports creating implicit transactions using the **TransactionScope** class.

[Implementing an Explicit Transaction using CommittableTransaction](http://msdn.microsoft.com/en-us/library/ms172146.aspx) describes how the **System.Transactions** namespace supports creating explicit transactions using the [CommittableTransaction](http://msdn.microsoft.com/en-us/library/system.transactions.committabletransaction.aspx) class.

For additional topics covering writing a transactional application, see [Writing a Transactional Application](http://msdn.microsoft.com/en-us/library/ms229973.aspx).

### Implementing A Resource Manager

To implement a resource manager that can participate in a transaction, see [Implementing a Resource Manager](http://msdn.microsoft.com/en-us/library/ms229975.aspx). This section covers the enlistment of a resource, committing a transaction, recovery after failure, and optimization best practices.

**Local Transactions (ADO.NET)**

Transactions in ADO.NET are used when you want to bind multiple tasks together so that they execute as a single unit of work. For example, imagine that an application performs two tasks. First, it updates a table with order information. Second, it updates a table that contains inventory information, debiting the items ordered. If either task fails, the both updates are rolled back.

 Determining the Transaction Type

A transaction considered to be a local transaction when it is a single-phase transaction and is handled by the database directly. Transactions are considered to be distributed transactions when they are coordinated by a transaction monitor and use fail-safe mechanisms (such as two-phase commit) for transaction resolution.

Each of the .NET Framework data providers has its own **Transaction** object for performing local transactions. If you require a transaction to be performed in a SQL Server database, select a [System.Data.SqlClient](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx) transaction. For an Oracle transaction, use the [System.Data.OracleClient](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.aspx) provider. In addition, there is a new [DbTransaction](http://msdn.microsoft.com/en-us/library/system.data.common.dbtransaction.aspx) class that is available for writing provider-independent code that requires transactions.

|  |
| --- |
| **NoteNote:** |
| Transactions are most efficient when it is performed on the server. If you are working with a SQL Server database that makes extensive use of explicit transactions, consider writing them as stored procedures using the Transact-SQL BEGIN TRANSACTION statement. For more information about performing server-side transactions, see SQL Server Books Online. |

 Performing a Transaction Using a Single Connection

In ADO.NET, you control transactions with the **Connection** object. You can initiate a local transaction with the **BeginTransaction** method. Once you have begun a transaction, you can enlist a command in that transaction with the **Transaction** property of a **Command** object. You can then commit or roll back modifications made at the data source based on the success or failure of the components of the transaction.

|  |
| --- |
| **NoteNote:** |
| The **EnlistDistributedTransaction** method should not be used for a local transaction. |

The scope of the transaction is limited to the connection. The following example performs an explicit transaction that consists of two separate commands in the **try** block. The commands execute INSERT statements against the Production.ScrapReason table in the AdventureWorks SQL Server 2005 sample database, which are committed if no exceptions are thrown. The code in the **catch** block rolls back the transaction if an exception is thrown. If the transaction is aborted or the connection is closed before the transaction has completed, it is automatically rolled back.

 Example

Follow these steps to perform a transaction.

1. Call the [BeginTransaction](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.begintransaction.aspx) method of the [SqlConnection](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx) object to mark the start of the transaction. The [BeginTransaction](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.begintransaction.aspx) method returns a reference to the transaction. This reference is assigned to the [SqlCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.aspx) objects that are enlisted in the transaction.
2. Assign the **Transaction** object to the [Transaction](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.transaction.aspx) property of the [SqlCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.aspx) to be executed. If a command is executed on a connection with an active transaction, and the **Transaction** object has not been assigned to the **Transaction** property of the **Command** object, an exception is thrown.
3. Execute the required commands.
4. Call the [Commit](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqltransaction.commit.aspx) method of the [SqlTransaction](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqltransaction.aspx) object to complete the transaction, or call the [Rollback](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqltransaction.rollback.aspx) method to end the transaction. If the connection is closed or disposed before either the [Commit](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqltransaction.commit.aspx) or [Rollback](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqltransaction.rollback.aspx) methods have been executed, the transaction is rolled back.

The following code example demonstrates transactional logic using ADO.NET with Microsoft SQL Server.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl28_ctl00_ctl11_code');" \o "Copy Code)

Using connection As New SqlConnection(connectionString)

connection.Open()

' Start a local transaction.

Dim sqlTran As SqlTransaction = connection.BeginTransaction()

' Enlist a command in the current transaction.

Dim command As SqlCommand = connection.CreateCommand()

command.Transaction = sqlTran

Try

' Execute two separate commands.

command.CommandText = \_

"INSERT INTO Production.ScrapReason(Name) VALUES('Wrong size')"

command.ExecuteNonQuery()

command.CommandText = \_

"INSERT INTO Production.ScrapReason(Name) VALUES('Wrong color')"

command.ExecuteNonQuery()

' Commit the transaction

sqlTran.Commit()

Console.WriteLine("Both records were written to database.")

Catch ex As Exception

' Handle the exception if the transaction fails to commit.

Console.WriteLine(ex.Message)

Try

' Attempt to roll back the transaction.

sqlTran.Rollback()

Catch exRollback As Exception

' Throws an InvalidOperationException if the connection

' is closed or the transaction has already been rolled

' back on the server.

Console.WriteLine(exRollback.Message)

End Try

End Try

End Using

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl28_ctl00_ctl12_code');" \o "Copy Code)

using (SqlConnection connection = new SqlConnection(connectionString))

{

connection.Open();

// Start a local transaction.

SqlTransaction sqlTran = connection.BeginTransaction();

// Enlist a command in the current transaction.

SqlCommand command = connection.CreateCommand();

command.Transaction = sqlTran;

try

{

// Execute two separate commands.

command.CommandText =

"INSERT INTO Production.ScrapReason(Name) VALUES('Wrong size')";

command.ExecuteNonQuery();

command.CommandText =

"INSERT INTO Production.ScrapReason(Name) VALUES('Wrong color')";

command.ExecuteNonQuery();

// Commit the transaction.

sqlTran.Commit();

Console.WriteLine("Both records were written to database.");

}

catch (Exception ex)

{

// Handle the exception if the transaction fails to commit.

Console.WriteLine(ex.Message);

try

{

// Attempt to roll back the transaction.

sqlTran.Rollback();

}

catch (Exception exRollback)

{

// Throws an InvalidOperationException if the connection

// is closed or the transaction has already been rolled

// back on the server.

Console.WriteLine(exRollback.Message);

}

}

}

**Distributed Transactions (ADO.NET)**

A transaction is a set of related tasks that either succeeds (commit) or fails (abort) as a unit, among other things. A *distributed transaction* is a transaction that affects several resources. For a distributed transaction to commit, all participants must guarantee that any change to data will be permanent. Changes must persist despite system crashes or other unforeseen events. If even a single participant fails to make this guarantee, the entire transaction fails, and any changes to data within the scope of the transaction are rolled back.

|  |
| --- |
| **NoteNote:** |
| An exception will be thrown if you attempt to commit or roll back a transaction if a **DataReader** is started while the transaction is active. |

 Working with System.Transactions

In the .NET Framework, distributed transactions are managed through the API in the [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) namespace. The [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) API will delegate distributed transaction handling to a transaction monitor such as the Microsoft Distributed Transaction Coordinator (MS DTC) when multiple persistent resource managers are involved. For more information, see [Transaction Fundamentals](http://msdn.microsoft.com/en-us/library/z80z94hz.aspx).

ADO.NET 2.0 introduced support for enlisting in a distributed transaction using the **EnlistTransaction** method, which enlists a connection in a [Transaction](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.aspx) instance. In previous versions of ADO.NET, explicit enlistment in distributed transactions was performed using the **EnlistDistributedTransaction** method of a connection to enlist a connection in a [ITransaction](http://msdn.microsoft.com/en-us/library/system.enterpriseservices.itransaction.aspx) instance, which is supported for backwards compatibility. For more information on Enterprise Services transactions, see [Interoperability with Enterprise Services and COM+ Transactions](http://msdn.microsoft.com/en-us/library/ms229974.aspx).

When using a [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) transaction with the .NET Framework Provider for SQL Server against a SQL Server 2005 database, a lightweight [Transaction](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.aspx) will automatically be used. The transaction can then be promoted to a full distributed transaction on an as-needed basis. For more information, see [System.Transactions Integration with SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/ms172070.aspx).

|  |
| --- |
| **NoteNote:** |
| The maximum number of distributed transactions that an Oracle database can participate in at one time is set to 10 by default. After the 10th transaction when connected to an Oracle database, an exception is thrown. Oracle does not support **DDL** inside of a distributed transaction. |

 Automatically Enlisting in a Distributed Transaction

Automatic enlistment is the default (and preferred) way of integrating ADO.NET connections with **System.Transactions**. A connection object will automatically enlist in an existing distributed transaction if it determines that a transaction is active, which, in **System.Transaction** terms, means that **Transaction.Current** is not null. Automatic transaction enlistment occurs when the connection is opened. It will not happen after that even if a command is executed inside of a transaction scope. You can disable auto-enlistment in existing transactions by specifying Enlist=false as a connection string parameter for a [ConnectionString](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.connectionstring.aspx), or OLE DB Services=-7 as a connection string parameter for an [ConnectionString](http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbconnection.connectionstring.aspx). For more information on Oracle and ODBC connection string parameters, see [ConnectionString](http://msdn.microsoft.com/en-us/library/system.data.oracleclient.oracleconnection.connectionstring.aspx) and [ConnectionString](http://msdn.microsoft.com/en-us/library/system.data.odbc.odbcconnection.connectionstring.aspx).

 Manually Enlisting in a Distributed Transaction

If auto-enlistment is disabled or you need to enlist a transaction that was started after the connection was opened, you can enlist in an existing distributed transaction using the **EnlistTransaction** method of the [DbConnection](http://msdn.microsoft.com/en-us/library/system.data.common.dbconnection.aspx) object for the provider you are working with. Enlisting in an existing distributed transaction ensures that, if the transaction is committed or rolled back, modifications made by the code at the data source will be committed or rolled back as well.

Enlisting in distributed transactions is particularly applicable when pooling business objects. If a business object is pooled with an open connection, automatic transaction enlistment only occurs when that connection is opened. If multiple transactions are performed using the pooled business object, the open connection for that object will not automatically enlist in newly initiated transactions. In this case, you can disable automatic transaction enlistment for the connection and enlist the connection in transactions using **EnlistTransaction**.

**EnlistTransaction** takes a single argument of type [Transaction](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.aspx) that is a reference to the existing transaction. After calling the connection's **EnlistTransaction** method, all modifications made at the data source using the connection are included in the transaction. Passing a null value unenlists the connection from its current distributed transaction enlistment. Note that the connection must be opened before calling **EnlistTransaction**.

|  |
| --- |
| **NoteNote:** |
| Once a connection is explicitly enlisted on a transaction, it cannot be un-enlisted or enlisted in another transaction until the first transaction finishes. |
| **Caution noteCaution:** |
| **EnlistTransaction** throws an exception if the connection has already begun a transaction using the connection's [BeginTransaction](http://msdn.microsoft.com/en-us/library/system.data.common.dbconnection.begintransaction.aspx) method. However, if the transaction is a local transaction started at the data source (for example, executing the BEGIN TRANSACTION statement explicitly using a [SqlCommand](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.aspx)), **EnlistTransaction** will roll back the local transaction and enlist in the existing distributed transaction as requested. You will not receive notice that the local transaction was rolled back, and must manage any local transactions not started using [BeginTransaction](http://msdn.microsoft.com/en-us/library/system.data.common.dbconnection.begintransaction.aspx). If you are using the .NET Framework Data Provider for SQL Server (**SqlClient**) with SQL Server 2005, an attempt to enlist will throw an exception. All other cases will go undetected. |

 Promotable Transactions in SQL Server 2005

SQL Server 2005 supports promotable transactions in which a local lightweight transaction can be automatically promoted to a distributed transaction only if it is required. A promotable transaction does not invoke the added overhead of a distributed transaction unless the added overhead is required. For more information and a code sample, see [System.Transactions Integration with SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/ms172070.aspx).

 Configuring Distributed Transactions

You may need to enable the MS DTC over the network in order to use distributed transactions if you are using a newer operating system with the latest service packs applied, such as Windows XP or Windows Server 2003. If have the Windows Firewall enabled (the default for Windows XP Service Pack 2), you must allow the MS DTC service to use the network or open the MS DTC port.

**System.Transactions Integration with SQL Server (ADO.NET)**

The .NET Framework version 2.0 introduced a transaction framework that can be accessed through the [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) namespace. This framework exposes transactions in a way that is fully integrated in the .NET Framework, including ADO.NET.

In addition to the programmability enhancements, [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) and ADO.NET can work together to coordinate optimizations when you work with transactions. A promotable transaction is a lightweight (local) transaction that can be automatically promoted to a fully distributed transaction on an as-needed basis.

Starting with ADO.NET 2.0, [System.Data.SqlClient](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx) supports promotable transactions when you work with SQL Server 2005. A promotable transaction does not invoke the added overhead of a distributed transaction unless the added overhead is required. Promotable transactions are automatic and require no intervention from the developer.

Promotable transactions are only available when you use the .NET Framework Data Provider for SQL Server (**SqlClient**) with SQL Server 2005.

 Creating Promotable Transactions

The .NET Framework Provider for SQL Server provides support for promotable transactions, which are handled through the classes in the .NET Framework [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) namespace. Promotable transactions optimize distributed transactions by deferring creating a distributed transaction until it is needed. If only one resource manager is required, no distributed transaction occurs.

|  |
| --- |
| **NoteNote:** |
| In a partially trusted scenario, the [DistributedTransactionPermission](http://msdn.microsoft.com/en-us/library/system.transactions.distributedtransactionpermission.aspx) is required when a transaction is promoted to a distributed transaction. For more information, see [Transaction Management Escalation](http://msdn.microsoft.com/en-us/library/ms229978.aspx). |

 Promotable Transaction Scenarios

Distributed transactions typically consume significant system resources, being managed by Microsoft Distributed Transaction Coordinator (MS DTC), which integrates all the resource managers accessed in the transaction. A promotable transaction is a special form of a [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) transaction that effectively delegates the work to a simple SQL Server 2005 transaction. [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx), [System.Data.SqlClient](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx), and SQL Server 2005 coordinate the work involved in handling the transaction, promoting it to a full distributed transaction as needed.

The benefit of using promotable transactions is that when a connection is opened by using an active [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) transaction, and no other connections are opened, the transaction commits as a lightweight transaction, instead of incurring the additional overhead of a full distributed transaction.

 Promotable Transactions in SQL Server 2008

In version 2.0 of the .NET Framework and SQL Server 2005, opening a second connection inside a [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) would automatically promote the transaction to a full distributed transaction, even if both connections were using identical connection strings. In this case, a distributed transaction adds unnecessary overhead that decreases performance.

Starting with SQL Server 2008 and version 3.5 of the .NET Framework, local transactions are no longer promoted to distributed transactions if another connection is opened in the transaction after the previous transaction is closed. This requires no changes to your code if you are already using connection pooling and enlisting in transactions. The following sections describe the situation in more detail.

**Understanding Lightweight Transactions**

A lightweight transaction is an agreement between SqlClient and the transaction, where **SqlClient** manages transacting the work on the transaction’s behalf. **SqlClient** starts a local server transaction when enlisting into the transaction, and the transaction sends a request to roll back or commit the work at the end.

When a second resource tries to enlist in the transaction using the same connection, the local transaction manager does not detect that there is already a connection enlisted to the same data source. In this situation, the local transaction promotes to a full distributed transaction.

Even though both connections are connecting to the same data source, connection pooling will not enable you to open another connection and enlist in the same transaction.

**Understanding Connection Pooling and Transactions**

A [SqlConnection](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx) consists of two parts: the public instance that your code interacts with (the outer connection) and a hidden connection that represents an actual server connection (the inner connection).

When you call the [Open](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.open.aspx) method on the outer connection, the connection manager looks for a free inner connection from the pool that is associated with the transaction. It will create a new inner connection if one does not exist. When the [Close](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.close.aspx) method is called on the outer connection, it returns the inner connection to the pool, where the connection [State](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.state.aspx) is reset to ensure a clean slate when it is reused from the pool. SQL Server does not distinguish between the outer and inner connections, so only one outer connection can be used with one inner connection at a time.

When a pooled connection is reset, cursors are closed, options are set back to defaults, the database context is switched back to the one specified in the connection string, and all outstanding local transactions are rolled back. This behavior provides a known starting state every time that you open a connection from the pool. It also prevents you from reusing the inner connection until the local transaction finishes. When you close the transaction's outer connection, the inner connection is set aside pending a commit or rollback request from the transaction manager. When you open a new outer connection while the inner connection is still active, the pool will be empty, and a new inner connection is created. The local transaction manager will then detect that two different resources are trying to enlist in the same transaction. Because SQL Server connections cannot share in local transactions, the transaction manager must promote the transaction to a full distributed transaction.

**Extensions Added for SQL Server 2008**

SQL Server 2008 adds a new connection reset mode that does not roll back local transactions. This enables **SqlClient** to return the inner connection to the pool to be reused, even when the connection is associated with a lightweight transaction. A second connection to the same data source can be associated with the existing local transaction without the transaction manager detecting a second transaction.

However, if you try to open a second outer transaction before closing the first connection, there will not be a free connection in the pool. A second inner connection will then be enlisted, and the transaction will be promoted. If connection pooling is turned off, or if a second connection uses a slightly different connection string, the inner connection will be unavailable even if it is sitting idle in the pool. For more information about SQL Server connection pooling, see [SQL Server Connection Pooling (ADO.NET)](http://msdn.microsoft.com/en-us/library/8xx3tyca.aspx).

No changes to your code are necessary in order to take advantage of this new functionality in SQL Server 2008. However, if you are enlisting in SQL Server 2005 transactions, pooling behavior reverts back to version 2.0 of the .NET Framework.

 The Enlist and Transaction Binding Connection String Keywords

The [ConnectionString](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.connectionstring.aspx) property supports a keyword, **Enlist**, which indicates whether [System.Data.SqlClient](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx) will detect transactional contexts and automatically enlist the connection in a distributed transaction. If Enlist=true, the connection is automatically enlisted in the opening thread's current transaction context. If Enlist=false, the **SqlClient** connection does not interact with a distributed transaction. The default value for **Enlist** is true. If **Enlist** is not specified in the connection string, the connection is automatically enlisted in a distributed transaction if one is detected when the connection is opened.

The **Transaction Binding** keywords in a [SqlConnection](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx) connection string control the connection's association with an enlisted **System.Transactions** transaction. It is also available through the [TransactionBinding](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnectionstringbuilder.transactionbinding.aspx) property of a [SqlConnectionStringBuilder](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnectionstringbuilder.aspx).

The following table describes the possible values.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| Implicit Unbind | The default. The connection detaches from the transaction when it ends, switching back to autocommit mode. |
| Explicit Unbind | The connection remains attached to the transaction until the connection is closed or **EnlistTransaction**(null) is called. The connection will fail if the associated transaction is not active or does not match [Current](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.current.aspx). |

For more information, see [Implementing an Implicit Transaction using Transaction Scope](http://msdn.microsoft.com/en-us/library/ms172152.aspx).

 Using TransactionScope

The [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) class makes a code block transactional by implicitly enlisting connections in a distributed transaction. You must call the [Complete](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.complete.aspx) method at the end of the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) block before leaving it. Leaving the block invokes the [Dispose](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.dispose.aspx) method. If an exception has been thrown that causes the code to leave scope, the transaction is considered aborted.

We recommend that you use a **using** block to make sure that [Dispose](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.dispose.aspx) is called on the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) object when the using block is exited. Failure to commit or roll back pending transactions can significantly damage performance because the default time-out for the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) is one minute. If you do not use a **using** statement, you must perform all work in a **Try** block and explicitly call the [Dispose](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.dispose.aspx) method in the **Finally** block.

If an exception occurs in the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx), the transaction is marked as inconsistent and is abandoned. It will be rolled back when the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) is disposed. If no exception occurs, participating transactions commit.

|  |
| --- |
| **NoteNote:** |
| The **TransactionScope** class creates a transaction with a [IsolationLevel](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.isolationlevel.aspx) of **Serializable** by default. Depending on your application, you might want to consider lowering the isolation level to avoid high contention in your application. |
| **NoteNote:** |
| We recommend that you perform only updates, inserts, and deletes within distributed transactions because they consume significant database resources. Select statements may lock database resources unnecessarily, and in some scenarios, you may have to use transactions for selects. Any non-database work should be done outside the scope of the transaction, unless it involves other transacted resource managers. Although an exception in the scope of the transaction prevents the transaction from committing, the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) class has no provision for rolling back any changes your code has made outside the scope of the transaction itself. If you have to take some action when the transaction is rolled back, you must write your own implementation of the [IEnlistmentNotification](http://msdn.microsoft.com/en-us/library/system.transactions.ienlistmentnotification.aspx) interface and explicitly enlist in the transaction. |

 Example

Working with [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) requires that you have a reference to System.Transactions.dll.

The following function demonstrates how to create a promotable transaction against two different SQL Server instances, represented by two different [SqlConnection](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx) objects, which are wrapped in a [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) block. The code creates the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) block with a **using** statement and opens the first connection, which automatically enlists it in the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx). The transaction is initially enlisted as a lightweight transaction, not a full distributed transaction. The second connection is enlisted in the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) only if the command in the first connection does not throw an exception. When the second connection is opened, the transaction is automatically promoted to a full distributed transaction. The [Complete](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.complete.aspx) method is invoked, which commits the transaction only if no exceptions have been thrown. If an exception has been thrown at any point in the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) block, **Complete** will not be called, and the distributed transaction will roll back when the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) is disposed at the end of its **using** block.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl68_ctl00_ctl09_code');" \o "Copy Code)

' This function takes arguments for the 2 connection strings and commands in order

' to create a transaction involving two SQL Servers. It returns a value > 0 if the

' transaction committed, 0 if the transaction rolled back. To test this code, you can

' connect to two different databases on the same server by altering the connection string,

' or to another RDBMS such as Oracle by altering the code in the connection2 code block.

Public Function CreateTransactionScope( \_

ByVal connectString1 As String, ByVal connectString2 As String, \_

ByVal commandText1 As String, ByVal commandText2 As String) As Integer

' Initialize the return value to zero and create a StringWriter to display results.

Dim returnValue As Integer = 0

Dim writer As System.IO.StringWriter = New System.IO.StringWriter

' Create the TransactionScope in which to execute the commands, guaranteeing

' that both commands will commit or roll back as a single unit of work.

Using scope As New TransactionScope()

Using connection1 As New SqlConnection(connectString1)

Try

' Opening the connection automatically enlists it in the

' TransactionScope as a lightweight transaction.

connection1.Open()

' Create the SqlCommand object and execute the first command.

Dim command1 As SqlCommand = New SqlCommand(commandText1, connection1)

returnValue = command1.ExecuteNonQuery()

writer.WriteLine("Rows to be affected by command1: {0}", returnValue)

' If you get here, this means that command1 succeeded. By nesting

' the Using block for connection2 inside that of connection1, you

' conserve server and network resources by opening connection2

' only when there is a chance that the transaction can commit.

Using connection2 As New SqlConnection(connectString2)

Try

' The transaction is promoted to a full distributed

' transaction when connection2 is opened.

connection2.Open()

' Execute the second command in the second database.

returnValue = 0

Dim command2 As SqlCommand = New SqlCommand(commandText2, connection2)

returnValue = command2.ExecuteNonQuery()

writer.WriteLine("Rows to be affected by command2: {0}", returnValue)

Catch ex As Exception

' Display information that command2 failed.

writer.WriteLine("returnValue for command2: {0}", returnValue)

writer.WriteLine("Exception Message2: {0}", ex.Message)

End Try

End Using

Catch ex As Exception

' Display information that command1 failed.

writer.WriteLine("returnValue for command1: {0}", returnValue)

writer.WriteLine("Exception Message1: {0}", ex.Message)

End Try

End Using

' If an exception has been thrown, Complete will

' not be called and the transaction is rolled back.

scope.Complete()

End Using

' The returnValue is greater than 0 if the transaction committed.

If returnValue > 0 Then

writer.WriteLine("Transaction was committed.")

Else

' You could write additional business logic here, notify the caller by

' throwing a TransactionAbortedException, or log the failure.

writer.WriteLine("Transaction rolled back.")

End If

' Display messages.

Console.WriteLine(writer.ToString())

Return returnValue

End Function

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl68_ctl00_ctl10_code');" \o "Copy Code)

// This function takes arguments for the 2 connection strings and commands in order

// to create a transaction involving two SQL Servers. It returns a value > 0 if the

// transaction committed, 0 if the transaction rolled back. To test this code, you can

// connect to two different databases on the same server by altering the connection string,

// or to another RDBMS such as Oracle by altering the code in the connection2 code block.

static public int CreateTransactionScope(

string connectString1, string connectString2,

string commandText1, string commandText2)

{

// Initialize the return value to zero and create a StringWriter to display results.

int returnValue = 0;

System.IO.StringWriter writer = new System.IO.StringWriter();

// Create the TransactionScope in which to execute the commands, guaranteeing

// that both commands will commit or roll back as a single unit of work.

using (TransactionScope scope = new TransactionScope())

{

using (SqlConnection connection1 = new SqlConnection(connectString1))

{

try

{

// Opening the connection automatically enlists it in the

// TransactionScope as a lightweight transaction.

connection1.Open();

// Create the SqlCommand object and execute the first command.

SqlCommand command1 = new SqlCommand(commandText1, connection1);

returnValue = command1.ExecuteNonQuery();

writer.WriteLine("Rows to be affected by command1: {0}", returnValue);

// if you get here, this means that command1 succeeded. By nesting

// the using block for connection2 inside that of connection1, you

// conserve server and network resources by opening connection2

// only when there is a chance that the transaction can commit.

using (SqlConnection connection2 = new SqlConnection(connectString2))

try

{

// The transaction is promoted to a full distributed

// transaction when connection2 is opened.

connection2.Open();

// Execute the second command in the second database.

returnValue = 0;

SqlCommand command2 = new SqlCommand(commandText2, connection2);

returnValue = command2.ExecuteNonQuery();

writer.WriteLine("Rows to be affected by command2: {0}", returnValue);

}

catch (Exception ex)

{

// Display information that command2 failed.

writer.WriteLine("returnValue for command2: {0}", returnValue);

writer.WriteLine("Exception Message2: {0}", ex.Message);

}

}

catch (Exception ex)

{

// Display information that command1 failed.

writer.WriteLine("returnValue for command1: {0}", returnValue);

writer.WriteLine("Exception Message1: {0}", ex.Message);

}

}

// If an exception has been thrown, Complete will not

// be called and the transaction is rolled back.

scope.Complete();

}

// The returnValue is greater than 0 if the transaction committed.

if (returnValue > 0)

{

writer.WriteLine("Transaction was committed.");

}

else

{

// You could write additional business logic here, notify the caller by

// throwing a TransactionAbortedException, or log the failure.

writer.WriteLine("Transaction rolled back.");

}

// Display messages.

Console.WriteLine(writer.ToString());

return returnValue;

}

**Optimistic Concurrency (ADO.NET)**

In a multiuser environment, there are two models for updating data in a database: optimistic concurrency and pessimistic concurrency. The [DataSet](http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx) object is designed to encourage the use of optimistic concurrency for long-running activities, such as remoting data and interacting with data.

Pessimistic concurrency involves locking rows at the data source to prevent other users from modifying data in a way that affects the current user. 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. This model is primarily used in environments where there is heavy contention for data, so that the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

Therefore, in a pessimistic currency model, a user who updates a row 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 and causing records to be locked for relatively large periods of time.

|  |
| --- |
| **Note:** |
| If you need to update multiple rows in the same operation, then creating a transaction is a more scalable option than using pessimistic locking. |

By contrast, 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. Optimistic concurrency improves performance because 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. How the server resolves a concurrency violation is best shown by first describing the following example.

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 value in the database ("James") no longer matches the original value that User1 was expecting ("Bob"). The concurrency violation simply lets you know that the update failed. 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.

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:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl20_ctl00_ctl00_code');" \o "Copy Code)

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:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl20_ctl00_ctl01_code');" \o "Copy Code)

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.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl20_ctl00_ctl02_code');" \o "Copy Code)

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 causes the data to be 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.

**The DataAdapter.RowUpdated Event**

The **RowUpdated** event of the [DataAdapter](http://msdn.microsoft.com/en-us/library/system.data.common.dataadapter.aspx) object can be used in conjunction with the techniques described earlier, to provide notification to your application of optimistic concurrency violations. **RowUpdated** occurs after each attempt to update a **Modified** row from a **DataSet**. This enables you to add special handling code, including processing when an exception occurs, adding custom error information, adding retry logic, and so on. The [RowUpdatedEventArgs](http://msdn.microsoft.com/en-us/library/system.data.common.rowupdatedeventargs.aspx) object returns a **RecordsAffected** property containing the number of rows affected by a particular update command for a modified row in a table. By setting the update command to test for optimistic concurrency, the **RecordsAffected** property will, as a result, return a value of 0 when an optimistic concurrency violation has occurred, because no records were updated. If this is the case, an exception is thrown. The **RowUpdated** event enables you to handle this occurrence and avoid the exception by setting an appropriate **RowUpdatedEventArgs.Status** value, such as **UpdateStatus.SkipCurrentRow**. For more information about the **RowUpdated** event, see [Handling DataAdapter Events (ADO.NET)](http://msdn.microsoft.com/en-us/library/6d1wk41s.aspx).

Optionally, you can set **DataAdapter.ContinueUpdateOnError** to **true**, before calling **Update**, and respond to the error information stored in the **RowError** property of a particular row when the **Update** is completed. For more information, see [Handling DataRow and DataColumn Errors (ADO.NET)](http://msdn.microsoft.com/en-us/library/k3877412.aspx).

 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.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl21_ctl00_ctl00_code');" \o "Copy Code)

' Assumes connection is a valid SqlConnection.

Dim adapter As SqlDataAdapter = New SqlDataAdapter( \_

"SELECT CustomerID, CompanyName FROM Customers ORDER BY CustomerID", \_

connection)

' The Update command checks for optimistic concurrency violations

' in the WHERE clause.

adapter.UpdateCommand = New SqlCommand("UPDATE Customers " &

"(CustomerID, CompanyName) VALUES(@CustomerID, @CompanyName) " & \_

"WHERE CustomerID = @oldCustomerID AND CompanyName = " &

"@oldCompanyName", connection)

adapter.UpdateCommand.Parameters.Add( \_

"@CustomerID", SqlDbType.NChar, 5, "CustomerID")

adapter.UpdateCommand.Parameters.Add( \_

"@CompanyName", SqlDbType.NVarChar, 30, "CompanyName")

' Pass the original values to the WHERE clause parameters.

Dim parameter As SqlParameter = dataSet.UpdateCommand.Parameters.Add( \_

"@oldCustomerID", SqlDbType.NChar, 5, "CustomerID")

parameter.SourceVersion = DataRowVersion.Original

parameter = adapter.UpdateCommand.Parameters.Add( \_

"@oldCompanyName", SqlDbType.NVarChar, 30, "CompanyName")

parameter.SourceVersion = DataRowVersion.Original

' Add the RowUpdated event handler.

AddHandler adapter.RowUpdated, New SqlRowUpdatedEventHandler( \_

AddressOf OnRowUpdated)

Dim dataSet As DataSet = New DataSet()

adapter.Fill(dataSet, "Customers")

' Modify the DataSet contents.

adapter.Update(dataSet, "Customers")

Dim dataRow As DataRow

For Each dataRow In dataSet.Tables("Customers").Rows

If dataRow.HasErrors Then

Console.WriteLine(dataRow (0) & vbCrLf & dataRow.RowError)

End If

Next

Private Shared Sub OnRowUpdated( \_

sender As object, args As SqlRowUpdatedEventArgs)

If args.RecordsAffected = 0

args.Row.RowError = "Optimistic Concurrency Violation!"

args.Status = UpdateStatus.SkipCurrentRow

End If

End Sub

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl21_ctl00_ctl01_code');" \o "Copy Code)

// Assumes connection is a valid SqlConnection.

SqlDataAdapter adapter = new SqlDataAdapter(

"SELECT CustomerID, CompanyName FROM Customers ORDER BY CustomerID",

connection);

// The Update command checks for optimistic concurrency violations

// in the WHERE clause.

adapter.UpdateCommand = new SqlCommand("UPDATE Customers " +

(CustomerID, CompanyName) VALUES(@CustomerID, @CompanyName) " +

"WHERE CustomerID = @oldCustomerID AND CompanyName = " +

@oldCompanyName", connection);

adapter.UpdateCommand.Parameters.Add(

"@CustomerID", SqlDbType.NChar, 5, "CustomerID");

adapter.UpdateCommand.Parameters.Add(

"@CompanyName", SqlDbType.NVarChar, 30, "CompanyName");

// Pass the original values to the WHERE clause parameters.

SqlParameter parameter = adapter.UpdateCommand.Parameters.Add(

"@oldCustomerID", SqlDbType.NChar, 5, "CustomerID");

parameter.SourceVersion = DataRowVersion.Original;

parameter = adapter.UpdateCommand.Parameters.Add(

"@oldCompanyName", SqlDbType.NVarChar, 30, "CompanyName");

parameter.SourceVersion = DataRowVersion.Original;

// Add the RowUpdated event handler.

adapter.RowUpdated += new SqlRowUpdatedEventHandler(OnRowUpdated);

DataSet dataSet = new DataSet();

adapter.Fill(dataSet, "Customers");

// Modify the DataSet contents.

adapter.Update(dataSet, "Customers");

foreach (DataRow dataRow in dataSet.Tables["Customers"].Rows)

{

if (dataRow.HasErrors)

Console.WriteLine(dataRow [0] + "\n" + dataRow.RowError);

}

protected static void OnRowUpdated(object sender, SqlRowUpdatedEventArgs args)

{

if (args.RecordsAffected == 0)

{

args.Row.RowError = "Optimistic Concurrency Violation Encountered";

args.Status = UpdateStatus.SkipCurrentRow;

}

}