Using System.Transactions to write reliable applications (Lab 1)

(Visual C# - For Visual Basic go to page 13)

Objectives

After completing this lab, you will be able to:

a. Create TransactionScopes that use Transactions

b. Create nested TransactionScopes

c. Use transactions with transacted resources (transacted hash table and SQL database)

Prerequisites

Before working on this lab, you must have the following installed on your machine (installed by default on the Virtual PC image):

a. .NET Framework v2.0

b. Microsoft® Visual Studio® .NET 2005

c. SQL Server™ 2005 installed

Scenario

This demo introduces you to System.Transactions and the TransactionScope feature. It is meant to show how to create TransactionScopes with various TransactionScopeOptions, and the simplicity they provide for interacting with transacted resources.

Estimated time to complete this lab: 60 minutes

Notes:

The code for all these exercises is on your lab machine at **C:\Microsoft Hands On Labs\HOL204\TxLab1\CS\Completed Exercises**. If

you wish, you can open these files and cut and paste the code into your project instead of typing it.

Exercise 1

Create a TransactionScope and modify a Transacted Hash Table inside the scope

In this exercise, you will create console test application in Visual C# .NET that uses a TransactionScope and a hash table as a transacted resource. You will experience the power of using the transaction paradigm with memory based resource managers. You will use the Microsoft Visual Studio .NET Development Environment to create, compile, and run this lab.

Tasks	Detailed steps
1. Open project for Lab 1 in Visual Studio	a. Click **Start->Programs->Microsoft Visual Studio 2005 Beta 2->Microsoft Visual Studio 2005 Beta 2**. b. Click **File->Open->Project/Solution**. c. Navigate to **C:\Microsoft Hands On Labs\HOL204\TxLab1\CS\Exercises\Exercise1** and select the file **Lab1.sln**. d. Click **Open**
2. Add a reference to System.Transactions	Click **View->Solution Explorer**. In Solution Explorer, right-click on **References** under **Lab1** and click **Add Reference…** With the .NET tab selected, select the component name **System.Transactions.dll**. Click **OK**
3. Add a using statement for System.Transactions	a. Open TxLab1.cs by double clicking it in the Solution Explorer b. At the top of TxLab1.cs add the following: `using System.Transactions;`
4. Build your application	a. Select **Build->Build Solution**. b. If the build is successful, you will see the following in the output window: `Build: 1 succeeded or up-to-date, 0 failed, 0 skipped`

5. Add a new TransactionScope with a Using statement	a. Add the following TransactionScope to the Main method in TxLab1.cs
	`using (TransactionScope ts = new TransactionScope())`
	`{`
	`  Console.WriteLine("Completing TransactionScope.");`
	`  ts.Complete();`
	`}`

	Notes: As demonstrated in this sample, TransactionScope is the class to use to make transacted operations on resources. • Once the TransactionScope is instantiated (through the 'new' statement), every operation on a transacted resources will be part of that transaction. At that time, a transaction ambient context is created, and automatically picked up by the resource managers. There is no need to pass any transaction parameters to the resource managers. • Calling the 'Complete' method on the TransactionScope, instructs the Transaction Manager that the state across all resources is consistent, and the transaction can be committed. It is very good practice to put the call to 'Complete' as the last statement in the using block. Failing to call 'Complete' will abort the transaction. Therefore, if there are any failures or exception happening within the scope of transaction, the transaction will rollback and none of the operations will be committed. • The actual work of commit between the resources manager actually happens at the 'End Using' statement. At that point the TransactionManager will call the resource managers and tell them to either commit or rollback based on whether the 'Complete' method was called on the TransactionScope object. As we can see, by the combination of the TransactionScope object and the 'using' statement, it is very easy to specify a block of statements that should participate in the transaction. You should not worry whether the operation in the HashTable or the Database are successful or not. If there is no error, both operations are guaranteed to succeed and state will be committed, and if there is an error both operations will fail and the state will be rolled back to its original value.

6. Add a transacted hash table	**a.** Add a transacted hash table and modify the hash table inside the TransactionScope. PrintTxHT will print the contents of the hash table, and PauseForExit waits for the user to hit a key before exiting.
	
	Notes: As part of this lab, a transacted hash table is provided (TxHashTable). This class can be used within a transaction, and any operation on the hashtable will be persisted if the transaction commits, or rolled back if the transaction aborts.
	
	`TxHashTable transactedHash = new TxHashTable();`
	
	`using (TransactionScope ts = new TransactionScope())`
	`{`
	`Console.WriteLine("Adding values to hash table inside TransactionScope.");`
	`transactedHash.Add(1, "a");`
	`transactedHash.Add(2, "ab");`
	`transactedHash.Add(3, "abc");`
	
	`Console.WriteLine("\nHash table inside TransactionScope.");`
	`PrintTxHT(transactedHash);`

```
        Console.WriteLine("Completing TransactionScope.");
        ts.Complete();
}

Console.WriteLine("\nHash table after TransactionScope.");
PrintTxHT(transactedHash);

PauseForExit();
```

7. Add a try…catch around the TransactionScope

a. Add the following try…catch around the TransactionScope in the Main method in TxLab1.cs

```
TxHashTable transactedHash = new TxHashTable();

try
{
     using (TransactionScope ts = new TransactionScope())
     {
          Console.WriteLine("Adding values to hash table inside
TransactionScope.");
          transactedHash.Add(1, "a");
          transactedHash.Add(2, "ab");
          transactedHash.Add(3, "abc");

          Console.WriteLine("\nHash table inside
TransactionScope.");
          PrintTxHT(transactedHash);

          Console.WriteLine("Completing TransactionScope.");
          ts.Complete();
     }
}
catch (TransactionException)
{
     Console.WriteLine("Caught TransactionException from
TransactionScope.");
}
catch (ApplicationException appe)
{
     Console.WriteLine("Caught {0} from TransactionScope.",
appe.GetType());
}

Console.WriteLine("\nHash table after TransactionScope.");
PrintTxHT(transactedHash);
```

8. Add a dialog box that will allow an exception to be thrown from inside the TransactionScope

a. Add a dialog box to Main to prompt the user if they would like to Complete the TransactionScope or thrown an exception inside the scope

```
static void Main(string[] args)
{
     string Msg1 = "Would you like to complete the first scope?
Saying no will cause an exception to be thrown.";
     DialogResult completeScope1 = MessageBox.Show(Msg1,
"Complete Scope", MessageBoxButtons.YesNo);
```

b. Add the following if statement inside the scope before you call complete. If you select No in the dialog box, an exception will be thrown from the TransactionScope and the scope will not Complete. This will cause the transaction inside the scope to roll back.

	using (TransactionScope ts = new TransactionScope())
	{
	Console.WriteLine("Adding values to hash table inside TransactionScope.");
	transactedHash.Add(1, "a");
	transactedHash.Add(2, "ab");
	transactedHash.Add(3, "abc");
	
	Console.WriteLine("\nHash table inside TransactionScope.");
	PrintTxHT(transactedHash);
	
	if (completeScope1 == DialogResult.No)
	{
	Console.WriteLine("Throwing new exception from TransactionScope.");
	throw new ApplicationException("Exception from TransactionScope.");
	}
	Console.WriteLine("Completing TransactionScope.");
	ts.Complete();
	}

9. Build your application	a. Select **Build->Build Solution**. b. If the build is successful, you will see the following in the output window: Build: 2 succeeded or up-to-date, 0 failed, 0 skipped
10. Run the application	a. Click **Debug->Start Debugging** A dialog box will appear and if Yes is selected, Complete() will be called on the TransactionScope and the transaction will Commit. Since the transaction commits, the changes to the hash table will be reflected in the last hash table print. If No is selected, an exception will be thrown and the TransactionScope will not be completed. This will cause the transaction to abort and the changes made to the hash table will automatically be rolled back. In this case the hash table will be empty.

Conclusion

In this exercise you created a TransactionScope and inside of the scope used a transacted resource. The transacted hash table enlisted in the transaction automatically and committed or aborted the changes that were made to the table inside of the TransactionScope based on the outcome of the transaction. TransactionScopes make it easy to create transacted blocks of code. We also learned that throwing exceptions will roll back the transaction and the previous state of the hash table is restored.

Exercise 2

Add SQL as a resource in the TransactionScope

In this exercise, you will see how easy it is to work with SQL databases. You will add a connection to a local SQL database inside the TransactionScope and watch as SQL automatically enlists in the transaction for you. See how SQL commits or aborts the changes made inside of the TransactionScope for you when you complete or abort the scope.

Tasks	Detailed steps
1. Open project for Lab 1 in Visual Studio	**a.** Click **Start->Programs->Microsoft Visual Studio 2005 Beta 2->Microsoft Visual Studio 2005 Beta 2**. **b.** Click **File->Open->Project/Solution**. **c.** Navigate to **C:\Microsoft Hands On Labs\HOL204\TxLab1\CS\Exercises\Exercise2** and select the file **Lab1.sln**. **d.** Click **Open**
2. Add a reference to System.Data	**a.** Click **View->Solution Explorer**. **b.** In Solution Explorer, right-click on **References** under **Lab1** and click **Add Reference…** **c.** With the .NET tab selected, select the component name **System.Data** **d.** Click **OK**

3. Add a using statement for System.Data	**a.** Open TxLab1.cs by double clicking it in the Solution Explorer
	b. At the top of TxLab1.cs add the following:
	`using System.Data;`
	`using System.Data.SqlClient;`

4. Add commands to set up the SQL database	**a.** Add SQLMethods.cs to the Lab1 project by right clicking on **Lab1** in the Solution Explorer and selecting **Add->Existing Item** **b.** Navigate to **C:\Microsoft Hands On Labs\HOL204\TxLab1\CS\Exercises\Exercise2\Lab1** **c.** Select **SQLMethods.cs** and click **Add** **d.** Open up TxLab1.cs by double clicking it in the Solution Explorer **e.** Add the following methods to create a new database, create a table inside the database, set the value inside the table to 2, and print out the table it just created. `TxHashTable transactedHash = new TxHashTable();` **`SQLMethods.CreateDB();`**

```
SQLMethods.CreateSQLTable();
SQLMethods.PrintSQLTable();

try
{
    using (TransactionScope ts = new TransactionScope())
```

5. Add commands to set up the SQL database

a. Open a new connection to the to the SQL database inside the TransactionScope and update the table value to be 100. Since the connection was opened inside the TransactionScope, SQL enlists on the ambient transaction and the update to the database is transacted.

```
using (TransactionScope ts = new TransactionScope())
{
  Console.WriteLine("Adding values to hash table inside
TransactionScope.");
  transactedHash.Add(1, "a");
  transactedHash.Add(2, "ab");
  transactedHash.Add(3, "abc");

  Console.WriteLine("\nHash table inside TransactionScope.");
  PrintTxHT(transactedHash);

   using (SqlConnection cn = new SqlConnection("Initial Catalog
= TxDB;Server=.;Trusted_Connection=yes"))
   {
      cn.Open();

      Console.WriteLine("\nUpdating SQL value to be 100.");
      SqlCommand cmd = cn.CreateCommand();
      cmd.CommandText = "update t_txlab set c=100";
      cmd.ExecuteNonQuery();
    }

  if (completeScope1 == DialogResult.No)
  {
    Console.WriteLine("\nThrowing new exception from
TransactionScope.");
    throw new ApplicationException("Exception from
TransactionScope.");
  }
  Console.WriteLine("\nCompleting TransactionScope.");
  ts.Complete();
}
```

6. Add another call to print the contents of the SQL table after the scope ends

a. Add another call to print the contents of the SQL table after the scope ends so you can see if the table was updated.

```
Console.WriteLine("\nHash table after TransactionScope.");
PrintTxHT(transactedHash);

Console.WriteLine("\nSQL after TransactionScope.");
SQLMethods.PrintSQLTable();

PauseForExit();
```

7. Build your application	**a.** Select **Build**->**Build Solution**. **b.** If the build is successful, you will see the following in the output window: `Build: 2 succeeded or up-to-date, 0 failed, 0 skipped`
8. Run the application	**a.** Click **Debug->Start Debugging** `Try completing and aborting the TransactionScope and see how SQL rolls back the update when the transaction is rolled back. Inside the scope we update SQL to have the value 100. When the transaction commits, you should see that the value in SQL is still 100 at the end. If the transaction rolls back, you should see that SQL rolled back the update and the value is once again set to 2.`  `Note: If you see a print statement that says: "Warning trying to Create DB…it may already exist." Do not worry, this just means that the database it is trying to create is already there. It will not affect the example.`

Conclusion

In this exercise, we used the TransactionScope object and integrate two different transacted resources in a single transaction. We saw that upon completion of the transaction, the state of the hash table and the SQL database is always consistent – either both of them commit, or none of them commit.

Exercise 3

Create nested TransactionScopes

In this exercise, you will create a second "nested" TransactionScope using the TransactionScopeOptions and you will see how the scopes modify the ambient transaction. A scope created with:

TransactionScopeOption.Required - means that if there is an ambient transaction, it will use it; otherwise it will create a new transaction.

TransactionScopeOption.RequiresNew – means that the TransactionScope will create a new Transaction and that will be set as the ambient transaction inside the scope.

TransactionScopeOption.Suppress – means that there will not be any ambient transaction inside the TransactionScope.

Tasks	Detailed steps
1. Open project for Lab 1 in Visual Studio	a. Click **Start->Programs->Microsoft Visual Studio 2005 Beta 2->Microsoft Visual Studio 2005 Beta 2**. b. Click **File->Open->Project/Solution**. c. Navigate to **C:\Microsoft Hands On Labs\HOL204\TxLab1\CS\Exercises\Exercise3** and select the file **Lab1.sln**. d. Click **Open**

2. Add a second dialog box for the second TransactionScope	a. Open TxLab1.cs by double clicking **TxLab1.cs** in the Solution Explorer
	b. Add a dialog box to Main to prompt the user if they would like to Complete the second TransactionScope or thrown an exception inside the scope
	
	`static void Main(string[] args)`
	`{`
	`  string Msg1 = "Would you like to complete the first scope? Saying no will cause an exception to be thrown.";`
	`  DialogResult completeScope1 = MessageBox.Show(Msg1, "Complete Scope", MessageBoxButtons.YesNo);`
	
	`  string Msg2 = "Would you like to complete the second scope? Saying no will cause an exception to be thrown.";`
	`  DialogResult completeScope2 = MessageBox.Show(Msg2, "Complete Scope", MessageBoxButtons.YesNo);`

3. Add a new method that creates the second TransactionScope	a. Add a new method CreateSecondScope that will create another TransactionScope using TransactionScopeOption.RequiresNew
	
	`public static void CreateSecondScope(DialogResult completeScope2)`
	`{`
	`  try`

```
     {
          Console.WriteLine("\nCreating second TransactionScope
with RequiresNew.");
          using (TransactionScope ts2 = new
TransactionScope(TransactionScopeOption.RequiresNew))
          {

               if (completeScope2 == DialogResult.No)
               {
                    Console.WriteLine("Throwing new exception
from TransactionScope.");
                    throw new ApplicationException("Exception
from TransactionScope.");
               }

               Console.WriteLine("Completing
TransactionScope.");
               ts2.Complete();
          }
     }
     catch (TransactionException)
     {
          Console.WriteLine("Caught TransactionException from
TransactionScope.");
     }
     catch (ApplicationException appe)
     {
          Console.WriteLine("Caught {0} from TransactionScope.",
appe.GetType());
     }
}
```

Note that this TransactionScope is created with RequiresNew so it will create its own Transaction for the scope. This means that the work that is done inside the second TransactionScope will be not be affected by the first scope's Transaction. You can verify this at the end of the exercise by throwing an exception in the first scope and completing the second scope. The update done in the second scope should still be committed.

4. Move SQL update from Main to new method

a. Remove the update to SQL that is in the Main method and move it into the TransactionScope inside CreateSecondScope.

```
public static void CreateSecondScope(DialogResult
completeScope2)
{
     try
     {
          Console.WriteLine("\nCreating second TransactionScope
with Suppress.");
          using (TransactionScope ts2 = new
TransactionScope(TransactionScopeOption.RequiresNew))
          {

               using (SqlConnection cn = new
SqlConnection("Initial Catalog =
TxDB;Server=.;Trusted_Connection=yes"))
               {
                  cn.Open();

                  Console.WriteLine("\nUpdating SQL value to be
100.");
                  SqlCommand cmd = cn.CreateCommand();
                  cmd.CommandText = "update t_txlab set c=100";
                  cmd.ExecuteNonQuery();
               }


               if (completeScope2 == DialogResult.No)
```

	{
	Console.WriteLine("Throwing new exception from TransactionScope.");
	throw new ApplicationException("Exception from TransactionScope.");
	}

5. Add a call to CreateSecondScope	**a.** Add a call to the new method from inside of the TransactionScope in Main.
	
	`Console.WriteLine("\nHash table inside TransactionScope.");`
	`PrintTxHT(transactedHash);`
	
	`CreateSecondScope(completeScope2);`
	
	`if (completeScope1 == DialogResult.No)`
	`{`
	`  Console.WriteLine("Throwing new exception from TransactionScope.");`
	`  throw new ApplicationException("Exception from TransactionScope.");`
	`}`

6. Build your application	**a.** Select **Build**->**Build Solution**. **b.** If the build is successful, you will see the following in the output window: `Build: 2 succeeded or up-to-date, 0 failed, 0 skipped`
7. Run the application	**a.** Click **Debug->Start Debugging** Try completing the first scope and aborting the second scope (and vice versa). Notice that the work done in the second scope is completely independent of the Transaction in the outer scope because it has its own Transaction.

8. Modify TransactionScope to use Suppress.	**a.** Change TransactionScopeOption.RequiresNew to TransactionScopeOption.Suppress in the TransactionScope constructor for the second scope.
	
	`public static void CreateSecondScope(DialogResult completeScope2)`
	`{`
	`  try`
	`  {`
	`  Console.WriteLine("\nCreating second TransactionScope with RequiresNew.");`
	`  using (TransactionScope ts2 = new TransactionScope(TransactionScopeOption.`**`Suppress`**`))`
	`  {`
	
	b. Rebuild and Rerun the application (refer to steps 6 and 7)
	
	Creating a TransactionScope with Suppress means that there will not be an ambient Transaction inside of the scope. This means that all of the work done inside the scope will not be associated with any Transaction. In this example you will notice that the SQL update done inside the Suppress will persist no matter what happens to either of the TransactionScopes. Note: it is not necessary to throw an exception or call Complete on TransactionScopes created with Suppress since there is no ambient Transaction to Commit or

	Abort.

9. Modify TransactionScope to use Required.	**a.** Change TransactionScopeOption.Suppress to TransactionScopeOption.Required in the TransactionScope constructor for the second scope.
	
	`public static void CreateSecondScope(DialogResult completeScope2)`
	`{`
	`  try`
	`  {`
	`  Console.WriteLine("\nCreating second TransactionScope with RequiresNew.");`
	`  using (TransactionScope ts2 = new TransactionScope(TransactionScopeOption.`**`Required`**`))`
	`  {`
	
	b. Rebuild and Rerun the application (refer to steps 6 and 7)
	
	Creating a TransactionScope with Required means that the scope will use the ambient Transaction if there is one, otherwise it will create its own Transaction. In this case, the second scope will pick up the outer TransactionScope's Transaction and set that as the ambient transaction. In this example you will notice that the work done in both of the scopes is under the same transaction, so if either of the two scopes does not Complete, the transaction will abort and all the work for both scopes will be rolled back.

Conclusion

In this exercise, we learned how to create nested transaction scopes. We saw that a by default (or through the use of the Required option), a nested transaction scope will inherit the current transaction. As well, we saw that we can create an independent transaction scope, by using the ReqruiesNew option, that will commit or abort independently of the other transaction. We also learned what the different TransactionScopeOptions mean. RequiresNew is used if you require a new transaction to be created while Required means you need a transaction but do not care if it is new or not. Suppress is used if you explicitly do not want to have any transaction associated with a block of code. This is useful for operations you would like have happen even if the transaction rolls back. One possible example is logging.

Using System.Transactions to write reliable applications (Lab 1)

(Visual Basic- For Visual C# go to page 1)

Objectives

After completing this lab, you will be able to:

Create TransactionScopes that use Transactions

Create nested TransactionScopes

Use transactions with transacted resources (transacted hash table and SQL database)

Prerequisites

Before working on this lab, you must have the following installed on your machine (installed by default on the Virtual PC image):

a. .NET Framework v2.0

b. Visual Studio .NET 2005

c. SQL Server installed

Scenario

This demo introduces you to System.Transactions and the TransactionScope feature. It is meant to show how to create TransactionScopes with various TransactionScopeOptions, and the simplicity they provide for interacting with transacted resources.

Estimated time to complete this lab: 60 minutes

Note:

The code for all these exercises is on your lab machine at **C:\Microsoft Hands On Labs\HOL204\TxLab1\VB\Completed Exercises**. If you wish, you can open these files and cut and paste the code into your project instead of typing it.

Exercise 1

Create a TransactionScope and modify a Transacted Hash Table inside the scope

In this exercise, you will create console test application in Visual C# .NET that uses a TransactionScope and a hash table as a transacted resource. You will experience the power of using the transaction paradigm with memory based resource managers. You will use the Microsoft Visual Studio .NET Development Environment to create, compile, and run this lab.

Tasks	Detailed steps
1. Open project for Lab 1 in Visual Studio	Click **Start->Programs->Microsoft Visual Studio 2005 Beta 2->Microsoft Visual Studio 2005 Beta 2**. Click **File->Open->Project/Solution**. Navigate to **C:\Microsoft Hands On Labs\HOL204\TxLab1\VB\Exercises\Exercise1** and select the file **Lab1.sln**. Click **Open**
2. Add a reference to System.Transactions	Click **View->Solution Explorer**. In Solution Explorer, double-click on **MyProject** to open. In the main window, click on the **References** tab on the left. Click the **Add…** button With the .NET tab selected, select the component name **System.Transactions.dll**. Click **OK**
3. Add a using statement for System.Transactions	Open TxLab1.cs by double clicking it in the Solution Explorer At the top of Lab1.vb add the following: `Imports System.Transactions`
4. Build your application	Select **Build->Build Solution**. If the build is successful, you will see the following in the output window: `Build: 1 succeeded or up-to-date, 0 failed, 0 skipped`

5. Add a new TransactionScope with a Using statement	a. Add the following TransactionScope to the Main method in Lab1.vb
	`Using ts As TransactionScope = New TransactionScope()`
	`Console.WriteLine("Completing TransactionScope.")`
	`ts.Complete()`
	`End Using`
	Notes:
	As demonstrated in this sample, TransactionScope is the

Formatted: Bullets and Numbering

	class to use to make transacted operations on resources. • Once the TransactionScope is instantiated (through the 'new' statement), every operation on a transacted resources will be part of that transaction. At that time, a transaction ambient context is created, and automatically picked up by the resource managers. There is no need to pass any transaction parameters to the resource managers. • Calling the 'Complete' method on the TransactionScope, instructs the Transaction Manager that the state across all resources is consistent, and the transaction can be committed. It is very good practice to put the call to 'Complete' as the last statement in the using block. Failing to call 'Complete' will abort the transaction. Therefore, if there are any failures or exception happening within the scope of transaction, the transaction will rollback and none of the operations will be committed. • The actual work of commit between the resources manager actually happens at the 'End Using' statement. At that point the TransactionManager will call the resource managers and tell them to either commit or rollback based on whether the 'Complete' method was called on the TransactionScope object. As we can see, by the combination of the TransactionScope object and the 'using' statement, it is very easy to specify a block of statements that should participate in the transaction. You should not worry whether the operation in the HashTable or the Database are successful or not. If there is no error, both operations are guaranteed to succeed and state will be committed, and if there is an error both operations will fail and the state will be rolled back to its original value.

6. Add a transacted hash table	Add a transacted hash table and modify the hash table inside the TransactionScope. PrintTxHT will print the contents of the hash table, and PauseForExit waits for the user to hit a key before exiting.
	
	Notes: As part of this lab, a transacted hash table is provided (TxHashTable). This class can be used within a transaction, and any operation on the hashtable will be persisted if the transaction commits, or rolled back if the transaction aborts.
	
	`Dim transactedHash As New TransactedResources.TxHashTable()`
	`Using ts As TransactionScope = New TransactionScope()`
	`Console.WriteLine("Adding values to hash table inside TransactionScope.")`
	`transactedHash.Add(1, "a")`
	`transactedHash.Add(2, "ab")`
	`transactedHash.Add(3, "abc")`
	`Console.WriteLine("")`
	`Console.WriteLine("Hash table inside TransactionScope.")`
	`PrintTxHT(transactedHash)`
	
	`Console.WriteLine("Completing TransactionScope.")`
	`ts.Complete()`
	`End Using`

	Console.WriteLine("") **Console.WriteLine("Hash table after TransactionScope.")** **PrintTxHT(transactedHash)** PauseForExit()
7. Add a try…catch around the TransactionScope	Add the following Try…catch around the TransactionScope in the Main method in Lab1.vb Dim transactedHash As New TransactedResources.TxHashTable() **Try** Using ts As TransactionScope = New TransactionScope() Console.WriteLine("Adding values to hash table inside TransactionScope.") transactedHash.Add(1, "a") transactedHash.Add(2, "ab") transactedHash.Add(3, "abc") Console.WriteLine("") Console.WriteLine("Hash table inside TransactionScope.") PrintTxHT(transactedHash) Console.WriteLine("Completing TransactionScope.") ts.Complete() End Using **Catch ex As TransactionException** **Console.WriteLine("Caught TransactionException from TransactionScope.")** **Catch appe As ApplicationException** **Console.WriteLine("Caught {0} from TransactionScope.", appe.GetType())** **End Try** Console.WriteLine("") Console.WriteLine("Hash table after TransactionScope.") PrintTxHT(transactedHash) PauseForExit()
8. Add a dialog box that will allow an exception to be thrown from inside the TransactionScope	Add a dialog box to Main to prompt the user if they would like to Complete the TransactionScope or thrown an exception inside the scope Shared Sub Main(ByVal args() As String) **Dim Msg1 As String = "Would you like to complete the first scope? Saying no will cause an exception to " & _** **"be thrown."** **Dim completeScope1 As DialogResult = MessageBox.Show(Msg1, "Complete Scope", MessageBoxButtons.YesNo)** Add the following if statement inside the scope before you call complete. If you select No in the dialog box, an exception will be thrown from the TransactionScope and the scope will not Complete. This will cause the transaction inside the scope to roll back. Using ts As TransactionScope = New TransactionScope() Console.WriteLine("Adding values to hash table inside TransactionScope.") transactedHash.Add(1, "a") transactedHash.Add(2, "ab")

Formatted: Bullets and Numbering

Formatted: Bullets and Numbering

```
                    transactedHash.Add(3, "abc")
                    Console.WriteLine("")
                    Console.WriteLine("Hash table inside
TransactionScope.")
                    PrintTxHT(transactedHash)

                    If (completeScope1 = DialogResult.No) Then
                        Console.WriteLine("Throwing new
exception from TransactionScope.")
                        Throw New
ApplicationException("Exception from TransactionScope.")
                    End If

                    Console.WriteLine("Completing
TransactionScope.")
                    ts.Complete()
                End Using
```

9. Build your application

a. Select **Build**->**Build Solution**.

b. If the build is successful, you will see the following in the output window:

```
Build: 2 succeeded or up-to-date, 0 failed, 0 skipped
```

10. Run the application

Click **Debug->Start Debugging**

```
A dialog box will appear and if Yes is selected, Complete()
will be called on the TransactionScope and the transaction
will Commit.  Since the transaction commits, the changes to
the hash table will be reflected in the last hash table
print.  If No is selected, an exception will be thrown and
the TransactionScope will not be completed.  This will cause
the transaction to abort and the changes made to the hash
table will automatically be rolled back.  In this case the
hash table will be empty.
```

Formatted: Bullets and Numbering

Conclusion

In this exercise you created a TransactionScope and inside of the scope used a transacted resource. The transacted hash table enlisted in the transaction automatically and committed or aborted the changes that were made to the table inside of the TransactionScope based on the outcome of the transaction. TransactionScopes make it easy to create transacted blocks of code. We also learned that throwing exceptions will roll back the transaction and the previous state of the hash table is restored.

Exercise 2

Add SQL as a resource in the TransactionScope

In this exercise, you will see how easy it is to work with SQL databases. You will add a connection to a local SQL database inside the TransactionScope and watch as SQL automatically enlists in the transaction for you. See how SQL commits or aborts the changes made inside of the TransactionScope for you when you complete or abort the scope.

Tasks	Detailed steps
1. Open project for Lab 1 in Visual Studio	Click **Start->Programs->Microsoft Visual Studio 2005 Beta 2->Microsoft Visual Studio 2005 Beta 2**. Click **File->Open->Project/Solution**. Navigate to **C:\Microsoft Hands On Labs\HOL204\TxLab1\VB\Exercises\Exercise2** and select the file **Lab1.sln**. Click **Open**
2. Add a reference to System.Data	Click **View->Solution Explorer**. In Solution Explorer, double-click on **MyProject** to open. In the main window, click on the **References** tab on the left. Click the **Add…** button With the .NET tab selected, select the component name **System.Data** Click **OK**

3. Add a using statement for System.Data	**a.** Open TxLab1.cs by double clicking it in the Solution Explorer
	b. At the top of TxLab1.cs add the following:
	`Imports System.Data`
	`Imports System.Data.SqlClient`

4. Add commands to set up the SQL database	Add SQLMethods.vb to the Lab1 project by right clicking on **Lab1** in the Solution Explorer and selecting **Add->Existing Item**
	Navigate to **C:\Microsoft Hands On Labs\HOL204\TxLab1\VB\Exercises\Exercise2\Lab1**
	Select **SQLMethods.cs** and click **Add**
	Open up Lab1.vb by double clicking it in the Solution Explorer
	Add the following methods to create a new database, create a table inside the database, set the value inside the table to 2, and print out the table it just created.
	`Dim transactedHash As New TransactedResources.TxHashTable()`
	`SQLMethods.CreateDB()`
	`SQLMethods.CreateSQLTable()`
	`SQLMethods.PrintSQLTable()`
	`Try`
	`Using ts As TransactionScope = New TransactionScope()ts = new TransactionScope())`

Formatted: Bullets and Numbering

5. Add commands to set up the SQL database	Open a new connection to the to the SQL database inside the TransactionScope and update the table value to be 100. Since the connection was opened inside the TransactionScope, SQL enlists on the ambient transaction and the update to the database is transacted.
	
	`Using ts As TransactionScope = New TransactionScope()`
	`  Console.WriteLine("Adding values to hash table inside TransactionScope.")`
	`  transactedHash.Add(1, "a")`
	`  transactedHash.Add(2, "ab")`
	`  transactedHash.Add(3, "abc")`
	`  Console.WriteLine("")`
	`  Console.WriteLine("Hash table inside TransactionScope.")`
	`  PrintTxHT(transactedHash)`
	
	`  Using cn As New SqlConnection("Initial Catalog = TxDB;Server=.;Trusted_Connection=yes")`
	`  cn.Open()`
	`  Console.WriteLine("")`
	`  Console.WriteLine("Updating SQL value to be 100.")`
	`  Dim cmd As SqlCommand = cn.CreateCommand`
	`  cmd.CommandText = "update t_txlab set c=100"`
	`  cmd.ExecuteNonQuery()`
	`  End Using`
	
	`  If (completeScope1 = DialogResult.No) Then`
	`  Console.WriteLine("Throwing new exception from TransactionScope.")`
	`  Throw New ApplicationException("Exception from TransactionScope.")`
	`  End If`

6. Add another call to print the contents of the SQL table after the scope ends	**a.** Add another call to print the contents of the SQL table after the scope ends so you can see if the table was updated.
	
	`Console.WriteLine("")`
	`Console.WriteLine("Hash table after TransactionScope.")`
	`PrintTxHT(transactedHash)`
	
	`Console.WriteLine("")`
	`Console.WriteLine("SQL after TransactionScope.")`
	`SQLMethods.PrintSQLTable()`
	
	`PauseForExit()`

7. Build your application	Select **Build**->**Build Solution**. If the build is successful, you will see the following in the output window: `Build: 2 succeeded or up-to-date, 0 failed, 0 skipped`
8. Run the application	**b.** Click **Debug->Start Debugging** `Try completing and aborting the TransactionScope and see how SQL rolls back the update when the transaction is rolled back. Inside the scope we update SQL to have the value 100. When the transaction commits, you should see that the value in SQL is still 100 at the end. If the transaction rolls back, you should see that SQL rolled back the update and the value is once again set to 2.`  `Note: If you see a print statement that says: "Warning trying to Create DB…it may already exist." Do not worry, this just means that the database it is trying to create is already there. It will not affect the example.`

Conclusion

In this exercise, we used the TransactionScope object and integrate two different transacted resources in a single transaction. We saw that upon completion of the transaction, the state of the hash table and the SQL database is always consistent – either both of them commit, or none of them commit.

Exercise 3

Create nested TransactionScopes

In this exercise, you will create a second "nested" TransactionScope using the TransactionScopeOptions and you will see how the scopes modify the ambient transaction. A scope created with:

TransactionScopeOption.Required - means that if there is an ambient transaction, it will use it; otherwise it will create a new transaction.

TransactionScopeOption.RequiresNew – means that the TransactionScope will create a new Transaction and that will be set as the ambient transaction inside the scope.

TransactionScopeOption.Suppress – means that there will not be any ambient transaction inside the TransactionScope.

Tasks	Detailed steps
1. Open project for Lab 1 in Visual Studio	a. Click **Start->Programs->Microsoft Visual Studio 2005 Beta 2->Microsoft Visual Studio 2005 Beta 2**. b. Click **File->Open->Project/Solution**. c. Navigate to **C:\Microsoft Hands On Labs\HOL204\TxLab1\VB\Exercises\Exercise3** and select the file **Lab1.sln**. d. Click **Open**

2. Add a second dialog box for the second TransactionScope	a. Open Lab1.vb by double clicking **Lab1.vb** in the Solution Explorer
	b. Add a dialog box to Main to prompt the user if they would like to Complete the second TransactionScope or thrown an exception inside the scope
	
	`Shared Sub Main(ByVal args() As String)`
	
	`Dim Msg1 As String = "Would you like to complete the first scope? Saying no will cause an exception to " & _ "be thrown." Dim completeScope1 As DialogResult = MessageBox.Show(Msg1, "Complete Scope", MessageBoxButtons.YesNo)`
	
	`Dim Msg2 As String = "Would you like to complete the second scope? Saying no will cause an exception to" & _ " be thrown." Dim completeScope2 As DialogResult = MessageBox.Show(Msg2, "Complete Scope", MessageBoxButtons.YesNo)`

3. Add a new method that creates the second TransactionScope	Add a new method CreateSecondScope that will create another TransactionScope using TransactionScopeOption.RequiresNew **`Public Shared Sub CreateSecondScope(ByVal completeScope2 As DialogResult) Try`**

```
                Console.WriteLine("")
                Console.WriteLine("Creating second
TransactionScope with RequiresNew.")
                Using ts2 As New
TransactionScope(TransactionScopeOption.Required)


                    If (completeScope2 = DialogResult.No) Then
                        Console.WriteLine("Throwing new
exception from TransactionScope.")
                        Throw New
ApplicationException("Exception from TransactionScope.")
                    End If
                    Console.WriteLine("Completing
TransactionScope.")
                    ts2.Complete()
                End Using
            Catch ex As TransactionException
                Console.WriteLine("Caught TransactionException
from TransactionScope.")
            Catch appe As ApplicationException
                Console.WriteLine("Caught {0} from
TransactionScope.", appe.GetType())
            End Try
        End Sub
```

Note that this TransactionScope is created with RequiresNew so it will create its own Transaction for the scope. This means that the work that is done inside the second TransactionScope will be not be affected by the first scope's Transaction. You can verify this at the end of the exercise by throwing an exception in the first scope and completing the second scope. The update done in the second scope should still be committed.

4. Move SQL update from Main to new method

a. Remove the update to SQL that is in the Main method and move it into the TransactionScope inside CreateSecondScope.

```
            Try
                Console.WriteLine("")
                Console.WriteLine("Creating second
TransactionScope with RequiresNew.")
                Using ts2 As New
TransactionScope(TransactionScopeOption.Required)

                    Using cn As New SqlConnection("Initial
Catalog = TxDB;Server=.;Trusted_Connection=yes")
                        cn.Open()
                        Console.WriteLine("")
                        Console.WriteLine("Updating SQL value to
be 100.")
                        Dim cmd As SqlCommand = cn.CreateCommand
                        cmd.CommandText = "update t_txlab set
c=100"
                        cmd.ExecuteNonQuery()
                    End Using

                    If (completeScope2 = DialogResult.No) Then
                        Console.WriteLine("Throwing new
exception from TransactionScope.")
                        Throw New
ApplicationException("Exception from TransactionScope.")
                    End If
```

5. Add a call to CreateSecondScope

Add a call to the new method from inside of the TransactionScope in Main.

```
Console.WriteLine("Hash table inside TransactionScope.")
PrintTxHT(transactedHash)
```

	CreateSecondScope(completeScope2)
	
	`If (completeScope1 = DialogResult.No) Then`
	`  Console.WriteLine("Throwing new exception from TransactionScope.")`
	`  Throw New ApplicationException("Exception from TransactionScope.")`
	`End If`

6. Build your application	**a.** Select **Build**->**Build Solution**. **b.** If the build is successful, you will see the following in the output window: `Build: 2 succeeded or up-to-date, 0 failed, 0 skipped`
7. Run the application	Click **Debug->Start Debugging** Try completing the first scope and aborting the second scope (and vice versa). Notice that the work done in the second scope is completely independent of the Transaction in the outer scope because it has its own Transaction.

8. Modify TransactionScope to use Suppress.	**a.** Change TransactionScopeOption.RequiresNew to TransactionScopeOption.Suppress in the TransactionScope constructor for the second scope.
	
	`Public Shared Sub CreateSecondScope(ByVal completeScope2 As DialogResult)`
	`  Try`
	`  Console.WriteLine("")`
	`  Console.WriteLine("Creating second TransactionScope with RequiresNew.")`
	`  Using ts2 As New TransactionScope(TransactionScopeOption.`**`Suppress`**`)`
	
	b. Rebuild and Rerun the application (refer to steps 6 and 7)
	
	Creating a TransactionScope with Suppress means that there will not be an ambient Transaction inside of the scope. This means that all of the work done inside the scope will not be associated with any Transaction. In this example you will notice that the SQL update done inside the Suppress will persist no matter what happens to either of the TransactionScopes. Note: it is not necessary to throw an exception or call Complete on TransactionScopes created with Suppress since there is no ambient Transaction to Commit or Abort.

9. Modify TransactionScope to use Required.	**a.** Change TransactionScopeOption.Suppress to TransactionScopeOption.Required in the TransactionScope constructor for the second scope.
	
	`Public Shared Sub CreateSecondScope(ByVal completeScope2 As DialogResult)`
	`  Try`
	`  Console.WriteLine("")`
	`  Console.WriteLine("Creating second TransactionScope with RequiresNew.")`
	`  Using ts2 As New TransactionScope(TransactionScopeOption.`**`Required`**`)`

	b. Rebuild and Rerun the application (refer to steps 6 and 7) Creating a TransactionScope with Required means that the scope will use the ambient Transaction if there is one, otherwise it will create its own Transaction. In this case, the second scope will pick up the outer TransactionScope's Transaction and set that as the ambient transaction. In this example you will notice that the work done in both of the scopes is under the same transaction, so if either of the two scopes does not Complete, the transaction will abort and all the work for both scopes will be rolled back.

Conclusion

In this exercise, we learned how to create nested transaction scopes. We saw that a by default (or through the use of the Required option), a nested transaction scope will inherit the current transaction. As well, we saw that we can create an independent transaction scope, by using the ReqruiesNew option, that will commit or abort independently of the other transaction. We also learned what the different TransactionScopeOptions mean. RequiresNew is used if you require a new transaction to be created while Required means you need a transaction but do not care if it is new or not. Suppress is used if you explicitly do not want to have any transaction associated with a block of code. This is useful for operations you would like have happen even if the transaction rolls back. One possible example is logging.