Microsoft Tech·Ed 2005

SQL Server 2005: SQL Server and ADO.NET

Table of Contents

SQL Server 2005: SQL Server and ADO.NET

Objectives

NOTE: This lab focuses on the concepts in this module and as a result may not comply with Microsoft® security recommendations.

NOTE: The SQL Server™ 2005 labs are based on beta builds of the product. The intent of these labs is to provide you with a general feel of some of the planned features for the next release of SQL Server. As with all software development projects, the final version may differ from beta builds in both features and user interface. For the latest details on SQL Server 2005, please visit http://www.microsoft.com/sql/2005/.

After completing this lab, you will be able to:

- Build a Microsoft Windows® application displaying a bound data grid.
- Execute a long-running query that would normally block the user interface as it runs.
- Build a Windows application displaying a bound data grid.
- Use SqlDependency and SqlNotifications.

Scenario

This lab will show you how to use new functionality in ADO.NET 2.0 with SQL Server 2005. In the first exercise, you will learn how to utilize the asynchronous capabilities of ADO.NET 2.0 by building a User Interface that will allow continued user activity even when it is still servicing a long running query.

Prerequisites

- Basic programming experience with SQL Server
- Basic knowledge of either VB.NET or C# programming

Estimated Time to Complete This Lab

45 Minutes

Lab Setup

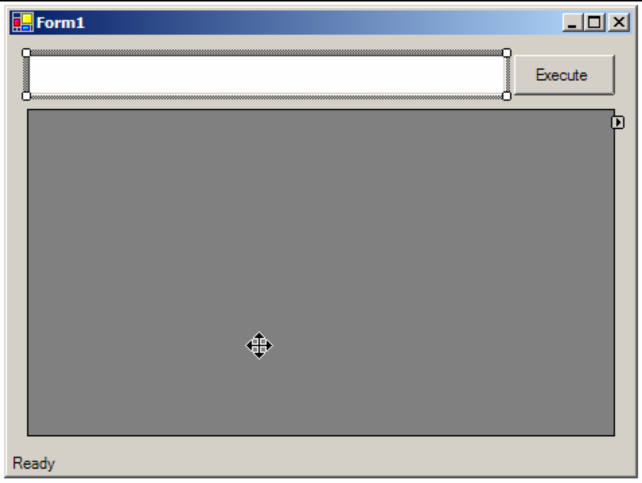
Tasks	Detailed Steps
1. Log in.	**1.** Log in using the **Administrator** user account. The password is **Pass@word1**.

Exercise 1
Building a Windows Form Application Utilizing the Asynchronous Capabilities of ADO.NET

Scenario

In this exercise, you will build an application that returns data and binds it to a DataGrid control. In order to fill the grid, you'll execute a command that simulates a long-running query. In order to demonstrate how you can avoid blocking the user interface while waiting for the results of the command, this demonstration uses the new asynchronous functionality available in ADO.NET. Although you could use any of the existing .NET asynchronous design patterns, this lab uses a delegate to illustrate this behavior.

Tasks	Detailed Steps
1. Create a Windows application.	**1.** From the Windows task bar, launch **Microsoft Visual Studio® 2005** by selecting **Start** | **All Programs** | **Microsoft Visual Studio 2005 Beta 2** | **Microsoft Visual Studio 2005 Beta 2**. **2.** From the menu select **File | Open | Project/Solution**. **3.** Create a new Windows application (in either C# or Microsoft Visual Basic® .NET). Set its name to **AdoNetAsync,** and its location to **C:\SQL Labs\User Projects.** **4.** In the Solution Explorer window, right-click the AdoNetAsync project, and select **Add Reference** from the context menu. In the Add Reference dialog box, select **System.Data.dll**, and click **OK** to add the reference. Your project may already include this reference, but adding it again won't cause any trouble. Repeat for the **System.Xml.dll** assembly.
2. Add controls to the form.	**1.** Use the **View | Toolbox** menu item to ensure that the Toolbox window is visible. Expand the **Windows Forms** tab in the toolbox, so you can use controls from this section. **2.** Add a TextBox control named **txtSql** to Form1. **3.** Add a DataGridView control named **grdDemo** to Form1. **4.** Add a Button control named **btnExecute** to Form1. Set the Text property to "Execute". **5.** Add a Label control named **lblInfo**. Set the label's AutoSize property to False. Dock the control to the bottom of the form. Set the Text property to "Ready". **6.** Lay out the controls on the form so that they resemble Figure 1.

Tasks	Detailed Steps
	 Figure 1: Completed form

3. Add Using/Import statements and a class level variable.	Now that you have created the form, you can add the code needed to retrieve data from the local SQL Server and bind it to the grid.
	1. Double-click btnExecute to load the code editor.
	2. Add a using/Imports statement for the System.Data.SqlClient namespace. In Visual Basic, add an Imports statement for the System.Data namespace, as well:
	`' Visual Basic`
	`Imports System.Data`
	`Imports System.Data.SqlClient`
	
	`// C#`
	`using System.Data.SqlClient;`
	
	3. Declare a private class-level SqlCommand object named cmd. If you're programming in C#, set its value to null:
	`' Visual Basic`
	`Private cmd As SqlCommand`
	
	`// C#`
	`private SqlCommand cmd = null;`

4. Add Synchronous data access code to the Form.	Add this code to the **Click Event** procedure of btnExecute.
	1. Declare a local SqlConnection variable named **cnn**, initialized to Nothing/null:
	`' Visual Basic`
	`Dim cnn As SqlConnection = Nothing`
	
	`// C#`
	`SqlConnection cnn = null;`

Tasks	Detailed Steps

	2. Add a Try/Catch block to the procedure. In the Catch block, catch an **Exception** object named **ex**.
	3. Until instructed otherwise, add code to the Try block. Inside the Try block, update **lblInfo** to display "Connecting…"
	`' Visual Basic`
	`lblInfo.Text = "Connecting ..."`
	`// C#`
	`lblInfo.Text = "Connecting ...";`
	4. Create and instantiate a SqlConnection variable cnn, using the connection string shown here:
	`' Visual Basic`
	`cnn = New SqlConnection( _`
	` "Data Source=localhost;Integrated Security=true;" & _`
	` "Initial Catalog=AdventureWorks")`
	`// C#`
	`cnn = new SqlConnection(`
	` "Data Source=localhost;Integrated Security=true;" +`
	` "Initial Catalog=AdventureWorks");`
	5. Open the connection:
	`' Visual Basic`
	`cnn.Open()`
	`// C#`
	`cnn.Open();`
	6. Update the text in lblInfo to display "Executing…":
	`' Visual Basic`
	`lblInfo.Text = "Executing ..."`
	`// C#`
	`lblInfo.Text = "Executing ...";`
	7. Create a new instance of the SqlCommand object, cmd, and in its constructor, specify the CommandText parameter to be the Text property of the txtSql textbox, and the connection to be the SqlConnection object you created earlier:
	`' Visual Basic`
	`cmd = New SqlCommand(txtSql.Text, cnn)`

Tasks	Detailed Steps

	`// C#`
	`cmd = new SqlCommand(txtSql.Text, cnn);`
	
	8. Create a new DataTable instance named dt:
	
	`' Visual Basic`
	`Dim dt As New DataTable()`
	
	`// C#`
	`DataTable dt = new DataTable();`
	
	9. Create a SqlDataReader and call the ExecuteReader method of the SqlCommand to supply its data:
	
	`' Visual Basic`
	`Dim reader As SqlDataReader = _`
	`  cmd.ExecuteReader(CommandBehavior.CloseConnection )`
	
	`// C#`
	`SqlDataReader reader =`
	`  cmd.ExecuteReader(CommandBehavior.CloseConnection);`
	
	10. Call the Load method of the DataTable, passing the SqlDataReader object as the parameter, and then close the SqlDataReader:
	
	`' Visual Basic`
	`dt.Load(reader)`
	`reader.Close()`
	
	`// C#`
	`dt.Load(reader);`
	`reader.Close();`
	
	11. Set the DataGrid control's DataSource property to be the DataTable you just filled:
	
	`' Visual Basic`
	`Me.grdDemo.DataSource = dt`
	
	`// C#`
	`this.grdDemo.DataSource = dt;`
	
	12. Update the text in lblInfo to say "Ready" again:
	
	`' Visual Basic`

Tasks	Detailed Steps

CAUTION: If you used C# code expansion to create the **try-catch-finally** code block, you must remove the **throw** statement. The MessageBox is used instead of a **throw** statement.	`lblInfo.Text = "Ready"`
	
	`// C#`
	`lblInfo.Text = "Ready";`
	
	13. Inside the Catch block, display the message corresponding to the exception:
	
	`' Visual Basic`
	`MessageBox.Show(ex.Message)`
	
	`// C#`
	`MessageBox.Show(ex.Message);`
	
	14. The completed procedure should look like the following:
	
	`' Visual Basic`
	`Dim cnn As SqlConnection = Nothing`
	
	`Try`
	`    lblInfo.Text = "Connecting..."`
	`    cnn = New SqlConnection( _`
	`     "Data Source=localhost;Integrated Security=true;" & _`
	`     "Initial Catalog=AdventureWorks")`
	`    cnn.Open()`
	
	`    lblInfo.Text = "Executing..."`
	`    cmd = New SqlCommand(Me.txtSql.Text, cnn)`
	`    Dim dt As New DataTable`
	`    Dim reader As SqlDataReader = _`
	`      cmd.ExecuteReader(CommandBehavior.CloseConnection)`
	`    dt.Load(reader)`
	`    reader.Close()`
	`    Me.grdDemo.DataSource = dt`
	`    lblInfo.Text = "Ready"`
	
	`Catch ex As Exception`
	`    MessageBox.Show(ex.Message)`
	`End Try`
	
	`// C#`
	`SqlConnection cnn = null;`
	
	`try`
	`{`
	`    lblInfo.Text = "Connecting...";`
	`    cnn = new SqlConnection(`
	`        "Data Source=localhost;Integrated Security=true;" +`
	`        "Initial Catalog=AdventureWorks");`
	`    cnn.Open();`

Tasks	**Detailed Steps**

	```
	    lblInfo.Text = "Executing...";
	    cmd = new SqlCommand(this.txtSql.Text, cnn);
	    DataTable dt = new DataTable();
	    SqlDataReader reader =
	     cmd.ExecuteReader(CommandBehavior.CloseConnection);
	    dt.Load(reader);
	    reader.Close();
	    this.grdDemo.DataSource = dt;
	    lblInfo.Text = "Ready";
	}
	catch (Exception ex)
	{
	    MessageBox.Show(ex.Message);
	}
	```
	 15. Run the application. When the form appears, enter the following query into the SQL Textbox: `WAITFOR DELAY '00:00:10'; SELECT * FROM Person.Contact` **16.** Click the **Execute** button.
	Note that while the query is executing the form's user interface will be unresponsive—try to resize the form, and your efforts will go unheeded until the query has completed its operation.

5. Add Asynchronous data access code to the Form.	In this step, you'll change the implementation and flow of the application. Instead of loading the data synchronously, this version of the application will use the asynchronous capabilities of ADO.NET to allow the form to remain responsive while waiting for results. You need to be aware of an important issue: when using the asynchronous capabilities of Windows applications, you must ensure that you bind the data to the grid on the form's thread, not on a background thread. Because of the way Windows GUI applications work, you may not interact with the form, its contents, or its properties from any thread besides the thread that created the form.
	In this example, much of the code you've already entered still applies but you'll need to change the flow of the code somewhat to accommodate the asynchronous features.
	1. Within the form's class, create a new Delegate to bind the data to the DataGrid:
	```
	' Visual Basic
	Private Delegate Sub UICallback(ByVal param As Object)
	```
	```
	// C#
	private delegate void UICallback(object param);
	```
	 You need to create two callback procedures for this application to work properly. One of the callbacks, ExecCallback, is called when the results from the query return. The other callback, ReBindOnUIThread, is needed when you interact with the user interface; for example, when you bind the results of your query to the grid. The important consideration here is that for Windows applications, any user interface interaction must be performed on the thread that created the form, and this

Tasks	Detailed Steps
	complicates the code.

2. Create a void method/sub named ReBindOnUIThread that has one parameter of type object. This method binds the resulting data to the grid. The code looks like the following snippet:

```
' Visual Basic
Private Sub ReBindOnUIThread(ByVal param As Object)
  If TypeOf param Is DataTable Then
    grdDemo.DataSource = param
    cmd = Nothing
    lblInfo.Text = "Ready"
  Else
    lblInfo.Text = "Ready (last failed: " & _
      CType(param, Exception).Message & ")"
  End If
End Sub
```

```
// C#
private void ReBindOnUIThread( object param )
{
  if( param is DataTable )
  {
    grdDemo.DataSource=param;
    cmd=null;
    lblInfo.Text="Ready";
  }
  else
    lblInfo.Text="Ready (last failed: " +
      ((Exception)param).Message + ")";
}
```

3. Create a void method/sub named ExecCallback with one parameter that is of type IAsyncResult. The code for this method looks like the following:

TIP: If Microsoft IntelliSense® does not display the Invoke method as a choice, click the **All** tab.

```
' Visual Basic
Private Sub ExecCallback(ByVal ar As IAsyncResult)
  Using reader As SqlDataReader = cmd.EndExecuteReader(ar)
    Try
      Dim tbl As New DataTable
      tbl.Load(reader)
      Me.Invoke( _
        New UICallback(AddressOf ReBindOnUIThread), _
        New Object() {tbl})
    Catch ex As Exception
      Me.Invoke( _
        New UICallback(AddressOf ReBindOnUIThread), _
        New Object() {ex})
      End Try
  End Using
End Sub
```

Tasks	Detailed Steps

	`// C#`
	`private void ExecCallback(IAsyncResult ar)`
	`{`
	`  using (SqlDataReader reader = cmd.EndExecuteReader(ar))`
	`  {`
	`    try`
	`    {`
	`      DataTable tbl = new DataTable();`
	`        tbl.Load(reader);`
	`      this.Invoke(new UICallback(ReBindOnUIThread),`
	`        new Object[] { tbl });`
	`    }`
	`    catch (Exception ex)`
	`    {`
	`      this.Invoke(new UICallback(ReBindOnUIThread),`
	`        new Object[] { ex });`
	`    }`
	`  }`
	`}`
	
	4. Modify the code in the button’s Click event so that instead of calling the ExecuteReader method, it calls the new BeginExecuteReader method. This method uses the standard .NET asynchronous callback design pattern, and allows you to pass a delegate that handles the work for you (in this case, the ExecCallback procedure). Remove the code that sets the label’s text property. You must also remove the code that creates and fills the DataTable, and then binds the DataTable to the DataGridView (that is, the remainder of the Try block after executing the reader)—that will happen in the callback procedure. Once you’re done, the call to BeginExecuteReader should look like the following:
	
	`' Visual Basic`
	`cmd.BeginExecuteReader( _`
	`   New AsyncCallback(AddressOf ExecCallback), Nothing, _`
	`  CommandBehavior.CloseConnection)`
	
	`// C#`
	`cmd.BeginExecuteReader(new AsyncCallback(ExecCallback ),`
	`  null, CommandBehavior.CloseConnection);`
	
	5. Modify the connection string, adding the **Asynchronous Processing=true** key/value pair. This feature allows asynchronous processing using the connection:
	
	`' Visual Basic`
	`cnn = New SqlConnection( _`
	`  "Data Source=.;Integrated Security=true;" & _`
	`  "Initial Catalog=AdventureWorks;" & _`
	`  "Asynchronous Processing=true")`

Tasks	Detailed Steps

```
// C#
cnn = new SqlConnection(
  "Data Source=.;Integrated Security=true;" +
  "Initial Catalog=AdventureWorks;" +
  "Asynchronous Processing=true");
```

6. In the catch block, add code to set the SqlCommand cmd to null/Nothing and if the SqlConnection object is not null/Nothing, close it:

```
' Visual Basic
cmd = Nothing
If cnn IsNot Nothing Then
  cnn.Close()
End If

// C#
cmd = null;
if (cnn != null)
{
  cnn.Close();
}
```

7. Make additional changes to the code so that the modified procedure looks like the following:

```
' Visual Basic
Dim cnn As SqlConnection = Nothing

Try
  lblInfo.Text = "Connecting..."
  cnn = New SqlConnection( _
    "Data Source=.;Integrated Security=true;" & _
    "Initial Catalog=AdventureWorks;" & _
    "Asynchronous Processing=true")
  cnn.Open()

  lblInfo.Text = "Executing..."
  cmd = New SqlCommand(Me.txtSql.Text, cnn)
  cmd.BeginExecuteReader(New AsyncCallback( _
    AddressOf ExecCallback), Nothing, _
    CommandBehavior.CloseConnection)

Catch ex As Exception
  MessageBox.Show(ex.Message)
  cmd = Nothing
  If cnn IsNot Nothing Then
    cnn.Close()
  End If
End Try
```

Tasks	Detailed Steps

	```
	// C#
	SqlConnection cnn = null;

	try
	{
	  lblInfo.Text = "Connecting...";
	  cnn = new SqlConnection(
	    "Data Source=.;Integrated Security=true;" +
	    "Initial Catalog=AdventureWorks;" +
	    "Asynchronous Processing=true"
	    );
	  cnn.Open();

	  lblInfo.Text = "Executing...";
	  cmd = new SqlCommand(this.txtSql.Text, cnn);
	  cmd.BeginExecuteReader(
	    new AsyncCallback(ExecCallback), null,
	    CommandBehavior.CloseConnection);
	}
	catch (Exception ex)
	{
	  MessageBox.Show(ex.Message);
	  cmd = null;
	  if (cnn != null)
	  {
	    cnn.Close();
	  }
	}
	```
	8. Save and build the application.
	9. Run the application. When the form appears, enter the following query into the Textbox control:
	`SELECT * FROM Person.Contact`
	10. Click **Execute**. The query returns and the data grid displays the bound data.
	11. Now enter the following query into the Textbox control. This query is intended to simulate a long-running operation. There could be other queries and/or actions that are occurring while the queries are being executed on the server side:
	`WAITFOR DELAY '00:00:10'; SELECT * FROM Person.Contact`
	12. Click **Execute**. Note that while the query is executing the user interface will still be responsive.

Exercise 2
Create a Windows Application Using SqlDependency

Scenario

The objective of this lab is to demonstrate the new SqlDependency and SqlNotifications infrastructure available from ADO.NET. These features provide the developer with the capability to execute queries and be notified when the specific data returned from that query has been changed. When coupled with ASP.NET, the entire response, or portions of the response, can be cached by the server and can provide greater performance and scalability.

You will create two applications for this exercise:

1.) A Windows application that binds data to a data grid.

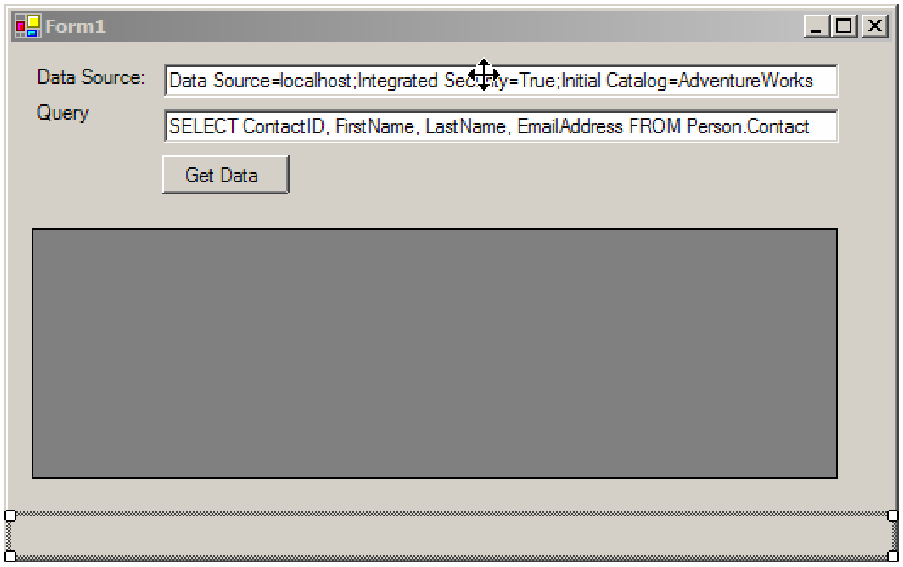
2.) A Windows application that adds, updates, and deletes a row in the database.

The first application will execute a command and register for a callback when the data has been retrieved. The second application will make changes to the same data so that SQL Server 2005 will notify the first Windows application.

Tasks	Detailed Steps

1. Build the Windows User Interface.	**1.** From the Windows task bar, launch **Visual Studio 2005** by selecting **Start | All Programs | Microsoft Visual Studio 2005 Beta 2 | Microsoft Visual Studio 2005 Beta 2**.
	2. From the menu select **File | Open | Project/Solution**.
	3. Create a new Windows application (in either C# or Visual Basic .NET). Set its name to **AdoNetDependency** and its location to **C:\SQL Labs\User Projects.**
	4. In the Solution Explorer window, right-click the **AdoNetDependency** project, and select **Add Reference** from the context menu. In the Add Reference dialog box, select **System.Data.dll**, and click OK to add the reference. Your project may already include this reference, but adding it again won't cause any trouble. Repeat for the **System.Xml.dll** assembly.
	5. From the **Windows Forms** tab of the Toolbox window, add a Label control to the form that's open in the form designer. Set its Text property to “Data Source:”
	6. Add a Textbox to the form to the right of the existing label, and set its Name property to **txtConnect**. Set its Text property to the following text:
	`Data Source=localhost;Integrated Security=True;Initial Catalog=AdventureWorks`
	7. Add a Label control below the first text box/label pair. Set its Text property to “Query:”
	8. Add a Textbox control to the right of the second label. Set the Name property to **txtSelect** and set its Text property to the following text:
	`SELECT ContactID, FirstName, LastName, EmailAddress FROM Person.Contact`
	9. Add a Button control beneath the two text box/label pairs. Set its Name property to **btnGetData** and its Text property to “Get Data”.
	10. Add a DataGridView control to the form. Set its Name property to **grdDemo**.
	11. Add a Label control to the bottom of the form. Set its AutoSize property to False.

Tasks	Detailed Steps

	Set its Name property to **lblStatus** and delete the text from its Text property. Set the label's Dock property so that it docks to the bottom of the form. Use the control's handles to give the control enough height to display status messages.
	12. The form should look similar to Figure 1.
	
	Data Source: Data Source=localhost;Integrated Security=True;Initial Catalog=AdventureWorks
	Query SELECT ContactID, FirstName, LastName, EmailAddress FROM Person.Contact
	Figure 1: The completed form.
	13. Double-click btnGetData to load the code editor.
	14. At the top of the form's code file, add the following imports/using statements:
	`' Visual Basic`
	`Imports System.Data`
	`Imports System.Data.SqlClient`
	
	`// C#`
	`using System.Data.SqlClient;`
	
	15. Inside the form's class, declare a class-level integer variable named **changeCount**, and instantiate a class-level DataSet variable named **ds**:
	`' Visual Basic`
	`Private changeCount As Integer`
	`Private ds As New DataSet`
	
	`// C#`
	`private int changeCount = 0;`
	`private DataSet ds = new DataSet();`

2. Create GetData method.	**1.** In the form's class, create a private/void sub/procedure with no parameters named **GetData**. **2.** Add a Try/Catch block to the new procedure. **3.** Inside the Try block

Tasks	Detailed Steps

NOTE: Don't worry that DataChanged appears as if it were a compile error. You'll create the DataChanged procedure in a later step.	a. Call the Clear method of the form's DataSet, ds.
	b. Create a new SqlConnection object named **connection** and set the ConnectionString property to txtConnect.Text.
	c. Create a new SqlDataAdapter object named **adapter** and in its constructor, pass txtSelect.Text and the connection object you created in the previous step.
	d. Create a new SqlDependency object named **dependency** and pass **adapter.SelectCommand** in its constructor.
	e. Specify that when the OnChange event of the SqlDependency object occurs, the code will call the DataChanged procedure in the form's class. The code looks like this:
	`' Visual Basic`
	`AddHandler dependency.OnChanged, _`
	`  AddressOf Me.DataChanged`
	
	`// C#`
	`dependency.OnChanged +=`
	`  new OnChangedEventHandler(this.DataChanged);`
	
	f. Call the **adapter.Fill** method, specifying the DataSet ds and the table "Contact".
	g. Set the DataSource property of the DataGridView control to the DataSet, ds. Set the DataMember property of the DataGridView to be the data table, "Contact".
	4. In the Catch block, display the exception message using the MessageBox.Show method.
	5. The completed procedure should look like the following:
	`' Visual Basic`
	`Private Sub GetData()`
	`  Try`
	`    ds.Clear()`
	`    Dim connection As New SqlConnection()`
	`    connection.ConnectionString = txtConnect.Text`
	
	`    Dim adapter As New SqlDataAdapter( _`
	`      txtSelect.Text, connection)`
	
	`    Dim dependency As New SqlDependency( _`
	`      adapter.SelectCommand)`
	
	`    ' Don't worry that DataChanged appears as if it were a`
	`    ' compile error. You'll create the DataChanged procedure`
	`    ' in a later step.`
	
	`    AddHandler dependency.OnChange, _`
	`      AddressOf Me.DataChanged`
	
	`    adapter.Fill(ds, "Contact")`

Tasks	Detailed Steps
	(continued below)

```
    grdDemo.DataSource = ds
    grdDemo.DataMember = "Contact"

  Catch ex As Exception
    MessageBox.Show(ex.Message)
  End Try
End Sub
```

```
//C#
private void GetData()
{
  try
  {
    ds.Clear();
    SqlConnection connection = new SqlConnection();
    connection.ConnectionString = txtConnect.Text;

    SqlDataAdapter adapter = new SqlDataAdapter(
      txtSelect.Text, connection);

    SqlDependency dependency =
      new SqlDependency(adapter.SelectCommand);

    // Don't worry that DataChanged appears as if it were a
    // compile error. You'll create the DataChanged
    // procedure in a later step.
    dependency.OnChange +=
      new OnChangedEventHandler(this.DataChanged);

    adapter.Fill(ds, "Contact");
    grdDemo.DataSource = ds;
    grdDemo.DataMember = "Contact";
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

Tasks	Detailed Steps
3. Create the callbacks to handle the SqlDependency notification and to bind data to the user interface.	**1.** In the form's class, create a Private/void Sub/method named ReBindOnUIThread with no parameters. This method will be invoked when the code receives the callback indicating that data has changed. **2.** In the ReBindOnUIThread procedure, call the GetData method and update the Label control on the form, so that the procedure looks like the following:

```
' Visual Basic
Private Sub ReBindOnUIThread()
  GetData()
  Me.lblStatus.Text = _
    String.Format("{0} changes have occurred.", _
    changeCount)
End Sub
```

Tasks	Detailed Steps

```
// C#
private void ReBindOnUIThread()
{
  GetData();
  this.lblStatus.Text =
    String.Format("{0} changes have occurred.",
    changeCount);
}
```

3. In the form's class, create a new Delegate that will bind the data to the grid:

```
' Visual Basic
Private Delegate Sub UICallback()
```

```
// C#
private delegate void UICallback();
```

4. In the form's class, create a callback procedure that will be invoked when the sample receives notification from the dependency object that the data has changed. The code should look like the following:

```
' Visual Basic
Private Sub DataChanged(ByVal sender As Object, _
 ByVal e As SqlNotificationEventArgs)

  changeCount += 1
  Me.Invoke(New UICallback(AddressOf ReBindOnUIThread))
End Sub
```

```
// C#
private void DataChanged(Object sender,
  SqlNotificationEventArgs e)
{
  changeCount++;
  this.Invoke(new UICallback(ReBindOnUIThread));
}
```

4. Add code to the btnGetData click event.

- Modify the btnGetData click event handler, adding the following code:

```
' Visual Basic
GetData()
Me.lblStatus.Text="No changes have occurred."
changeCount = 0
Me.btnGetData.Enabled = False
Me.txtSelect.Enabled = False
Me.txtConnect.Enabled = False
```

```
// C#
```

Tasks	**Detailed Steps**

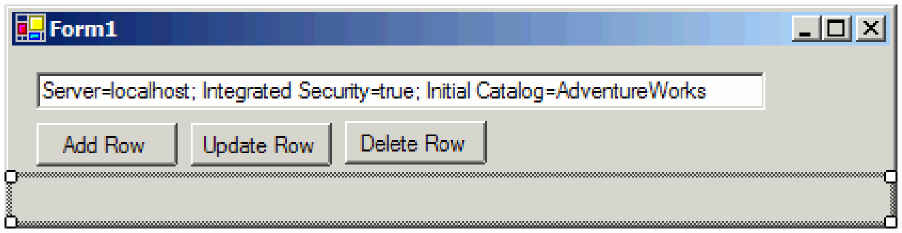
	`GetData();`
	`this.lblStatus.Text="No changes have occurred.";`
	`changeCount = 0;`
	`this.btnGetData.Enabled = false;`
	`this.txtSelect.Enabled = false;`
	`this.txtConnect.Enabled = false;`

5. Run the application.	**1.** Select the **File | Save All** menu command. **2.** Run the application, click the button, and verify that it fills the grid with data.

Exercise 3
Build the Change Application

Scenario

In this exercise, you will build an application that inserts, updates, and deletes a row of data in the Contact table.

Tasks	**Detailed Steps**
1. Build the User Interface.	**1.** Start a new instance of Visual Studio 2005. **2.** Create a new Windows application (in either C# or Visual Basic .NET). Set its name to **AdoNetChangeApp.** **3.** In the Solution Explorer window, right-click the **AdoNetChangeApp** project, and select **Add Reference** from the context menu. In the Add Reference dialog box, select **System.Data.dll**, and click OK to add the reference. Your project may already include this reference, but adding it again won't cause any trouble. Repeat for the **System.Xml.dll** assembly. **4.** Add a Textbox control to Form1. Set its Name property to **txtConnect** and its Text property to the following connection string: `Server=localhost; Integrated Security=true;Initial Catalog=AdventureWorks` **5.** Add a Label control to the form. Set the Label control's name to **lblStatus**. Set its AutoSize property to False, its Text property to an empty string. Set the control's Dock property so that it docks to the bottom of the form. Use the control’s handles to give the control enough height to display status messages. **6.** Add three Button controls to the form, setting properties as shown below: **Name** — **Text** btnAddRow — Add Row btnUpdateRow — Update Row btnDeleteRow — Delete Row Once you're done, the form should look something like Figure 1.  Server=localhost; Integrated Security=true; Initial Catalog=AdventureWorks Figure 1: The completed form.
2. Adding Change code to the Application.	**1.** Press **F7** to load the code editor. **2.** In the form's class, add an Imports/using statement to import the **System.Data.SqlClient** namespace. If you're using Visual Basic, add an Import statement to import the **System.Data** namespace, as well.

Tasks	Detailed Steps

	``' Visual Basic``
	 ``Imports System.Data.SqlClient``
	 ``Imports System.Data``
	
	 ``// C#``
	 ``using System.Data.SqlClient;``
	
	 3. Create a method/function named **ExecCommand** that returns an integer and accepts one string parameter named cmdText.
	
	 ``' Visual Basic``
	 ``Private Function ExecCommand( _``
	 ``  ByVal cmdText As String) As Integer``
	
	 ``End Function``
	
	 ``//C#``
	 ``private int ExecCommand(string cmdText)``
	 ``{``
	 ``}``
	
	 4. Inside the procedure, create a local variable named **result** that is of type int/Integer, and assign the variable the value 0.
	
	 ``' Visual Basic``
	 ``Dim result As Integer = 0``
	
	 ``//C#``
	 ``int result = 0;``
	
	 5. Inside the procedure, add a SqlConnection variable named **cnn**. In C#, initialize the variable to **null**.
	
	 ``' Visual Basic``
	 ``Dim cnn As SqlConnection``
	
	 ``//C#``
	 ``SqlConnection cnn = null;``
	
	 6. Add a Try/Catch block to the function. In the Catch block, catch an **Exception** object named **ex**.
	 7. Inside the Try block:
	 a. Instantiate cnn to be a new SqlConnection object and pass the constructor txtConnect.Text as the connection string.
	
	 ``' Visual Basic``

Tasks	Detailed Steps

	`cnn = New SqlConnection(txtConnect.Text)`
	
	`//C#`
	`cnn = new SqlConnection(txtConnect.Text);`
	
	b. Create a new SqlCommand object and pass the constructor cmdText (the parameter passed into this procedure) and the newly created SqlConnection object.
	
	`' Visual Basic`
	`Dim command As New SqlCommand( _`
	`  cmdText, cnn)`
	
	`//C#`
	`SqlCommand command =`
	`  new SqlCommand(cmdText, cnn);`
	
	c. Open the connection.
	
	`' Visual Basic`
	`cnn.Open()`
	
	`//C#`
	`cnn.Open();`
	
	d. Call the ExecuteNonQuery method of the SqlCommand object and assign the return value to the local variable named result.
	
	`' Visual Basic`
	`result = command.ExecuteNonQuery()`
	
	`//C#`
	`result = command.ExecuteNonQuery();`
	
	8. In the catch block, display the exception's message using MessageBox.Show.
	
	`' Visual Basic`
	`MessageBox.Show(ex.Message)`
	
	`//C#`
	`MessageBox.Show(ex.Message);`
	
	9. Add a Finally block, inserting code that checks to ensure that the connection is not null/nothing, and if so, closes the connection.
	
	`' Visual Basic`

Tasks	**Detailed Steps**

	`If cnn IsNot Nothing Then`
	`  cnn.Close()`
	`End If`
	
	`//C#`
	`if (cnn != null)`
	`  {`
	`  cnn.Close();`
	`  }`
	
	10. Return the **result** variable as the return value of the function.
	
	`' Visual Basic`
	`Return result`
	
	`//C#`
	`return result;`
	
	11. The completed code should look like the following:
	
	`' Visual Basic`
	`Private Function ExecCommand( _`
	`  ByVal cmdText As String) As Integer`
	
	`  Dim cnn As SqlConnection`
	`  Dim result As Integer = 0`
	
	`  Try`
	`    cnn = New SqlConnection(txtConnect.Text)`
	
	`    Dim command As New SqlCommand( _`
	`      cmdText, cnn)`
	
	`    cnn.Open()`
	`    result = command.ExecuteNonQuery()`
	
	`  Catch ex As Exception`
	`    MessageBox.Show(ex.Message)`
	
	`  Finally`
	`    If cnn IsNot Nothing Then`
	`      cnn.Close()`
	`    End If`
	`  End Try`
	`  Return result`
	`End Function`
	
	`//C#`
	`private int ExecCommand(string cmdText)`

Tasks	Detailed Steps

```
{
  int result = 0;
  SqlConnection cnn = null;

  try

  {
    cnn = new SqlConnection(txtConnect.Text);

    SqlCommand command =
      new SqlCommand(cmdText, cnn);

    cnn.Open();
    result = command.ExecuteNonQuery();
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
  finally
  {
    if (cnn != null)
    {
      cnn.Close();
    }
  }
    return result;
}
```

3. Creating click events for buttons.

1. Insert the following declaration into the form's class:

```
' Visual Basic
Private newID As Integer = -1
```

```
// C#
private int newID = -1;
```

2. Insert the following code into the **btnAddRow** Click event handler:

```
' Visual Basic
Using cnn As New SqlConnection(txtConnect.Text)
  Dim cmd As New SqlCommand()
  ' Make up some fake values:
  Dim g As Guid = Guid.NewGuid()
  cmd.CommandText = _
   "INSERT INTO Person.Contact " & _
   "(FirstName, LastName, NameStyle, " & _
   "PasswordHash, PasswordSalt, RowGuid, ModifiedDate) " & _
   "VALUES ('Ken', 'Smith', 0, 'XXXXX', 'XXXXX','" & _
    g.ToString() + "', '5/16/2005'); " & _
    "SELECT SCOPE_IDENTITY() AS ID"
```

Tasks	**Detailed Steps**

	`  cmd.Connection = cnn`
	`  cnn.Open()`
	`  newID = Convert.ToInt32(cmd.ExecuteScalar())`
	`End Using`
	
	`If newID > -1 Then`
	`  Me.lblStatus.Text = "Record Added."`
	`Else`
	`  Me.lblStatus.Text = "No Record Added."`
	`End If`
	
	`// C#`
	`using (SqlConnection cnn = new`
	`SqlConnection(txtConnect.Text))`
	`{`
	`  SqlCommand cmd = new SqlCommand();`
	`  // Make up some fake values:`
	`  Guid g = Guid.NewGuid();`
	`  cmd.CommandText =`
	`    "INSERT INTO Person.Contact " +`
	`    "(FirstName, LastName, NameStyle, " +`
	`    "PasswordHash, PasswordSalt, RowGuid, ModifiedDate) " +`
	`    "VALUES ('Ken', 'Smith', 0, 'XXXXX', 'XXXXX','" +`
	`    g.ToString() + "', '5/16/2005'); " +`
	`    "SELECT SCOPE_IDENTITY() AS ID";`
	`  cmd.Connection = cnn;`
	`  cnn.Open();`
	`  newID = Convert.ToInt32(cmd.ExecuteScalar());`
	`}`
	
	`if (newID > -1)`
	`  this.lblStatus.Text = "Record added.";`
	`else`
	`  this.lblStatus.Text = "Record not added.";`
	
	3. Insert the following code into the **btnUpdateRow** Click event handler:
	
	`' Visual Basic`
	`If newID > -1 Then`
	`  If ExecCommand( _`
	`    "UPDATE Person.Contact SET FirstName = 'Peter' " & _`
	`    "WHERE ContactID = " & newID.ToString()) > 0 Then`
	`    Me.lblStatus.Text = "Record Updated."`
	`  Else`
	`    Me.lblStatus.Text = "No Record Updated."`
	`  End If`
	`Else`
	`  MessageBox.Show( _`
	`    "Row cannot be updated before it is added.")`
	`End If`

Tasks	**Detailed Steps**

```
// C#
if (newID > -1)
{
  if (ExecCommand(
    "UPDATE Person.Contact SET FirstName = 'Peter' " +
    "WHERE ContactID = " + newID.ToString()) > 0)
    this.lblStatus.Text = "Record Updated.";
  else
    this.lblStatus.Text = "No Record Updated.";
}
else
  MessageBox.Show(
    "Row cannot be updated before it is added.");
```

4. Insert the following code into the **btnDeleteRow** Click event handler:

```
' Visual Basic
If newID > -1 Then
  If ExecCommand( _
   "DELETE FROM Person.Contact " & _
   "WHERE ContactID = " & newID.ToString()) > 0 Then
    Me.lblStatus.Text = "Record Deleted."
  Else
    Me.lblStatus.Text = "No Record Deleted."
  End If
  newID = -1
Else
  MessageBox.Show( _
   "Row cannot be deleted before it is added.")
End If
```

```
// C#
if (newID > -1)
{
  if (ExecCommand(
    "DELETE FROM Person.Contact " +
    "WHERE ContactID = " + newID.ToString()) > 0)
    this.lblStatus.Text = "Record Deleted.";
  else
    this.lblStatus.Text = "No Record Deleted.";
  newID = -1;
}
else
  MessageBox.Show(
  "Row cannot be deleted before it is added.");
```

Exercise 4
Enable Notifications and Test

Scenario

Before you can actually receive query notifications, the feature must be enabled on a per database base.

Tasks	Detailed Steps
1. Enable Notifications.	**1.** From the Windows task bar, select **Start | All Programs | Microsoft SQL Server 2005 | SQL Server Management Studio**. **2.** When the **Connect to Server** dialog box opens, verify that **SQL Server** is selected as the **Server type,** verify that **Server name** is same name as the host machine or set it to **localhost**, and verify that **Windows Authentication** is selected as the authentication method. **3.** Click **Connect**. **4.** Open a new query window by via **File | New | Database Engine Query** and enter the following T-SQL: `ALTER DATABASE AdventureWorks SET ENABLE_BROKER` **5.** Press **F5** to execute the query. **6.** Close SQL Server Management Studio (there's no need to save anything if prompted).
2. Test the Notification Applications.	**1.** Run the AdoNetDependency application. **2.** Click the **Get Data** button and note that the textboxes are read-only and the data is bound to the grid. **3.** Run the AdoNetChange application. Click the **Add** button and notice that the AdoNetDependency application registers the change and shows the new data. Scroll to the bottom of the grid to see the new record for Ken Smith. **4.** Click the **Update Row** button and notice, again, that the AdoNetDependency application receives a callback and shows the updated data. Scroll to the bottom of the grid to see that the Ken Smith record now says Peter Smith. **5.** Repeat for the **Delete** button.