DATABASE ACCESS IN VISUAL STUDIO - ADO.NET

The ActiveX Data Objects (ADO.NET) is part of the .NET framework that provides a multi-layered set of data access namespaces (set of classes), which enables access and modification of data stored in a database. To use these classes in the .NET Framework you must import the namespace `System.Data` that contains all these useful classes. The ADO.NET supports both connected and disconnected data architecture

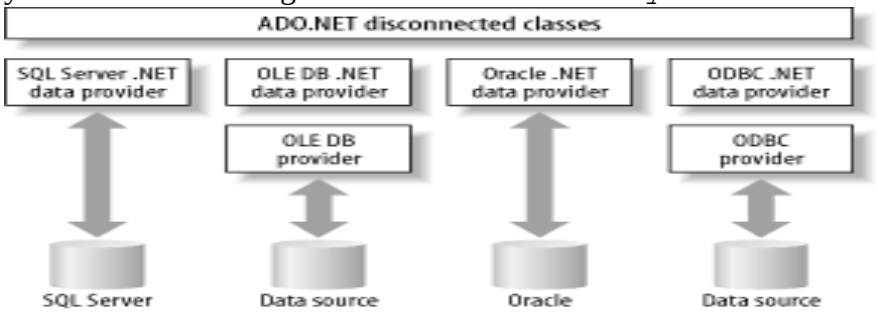
The ADO.Net Platforms

- Microsoft ACE Database Engine is the main Access-based component for storing, parsing, optimizing, and processing queries. ACE (Access Connectivity Engine -- now called the Access Database Engine) is the newest database engine for working with Access 2007 and later (".accdb" files) - successor to the Jet (Joint Engine Technology) model. It is a File-Server System DBMS where the work is done on the individual computer where the Access DBMS exists.
- Microsoft SQL Server is a client-server DBMS in which the client requests information from the database and the server processes the request i.e. it provides functionalities beyond those available through JET or ACE such as improved multi-user access and transaction-based processing.
- Oracle is another client-server DBMS used for large databases.

ADO.NET Libraries

ADO.NET supports multiple database platforms

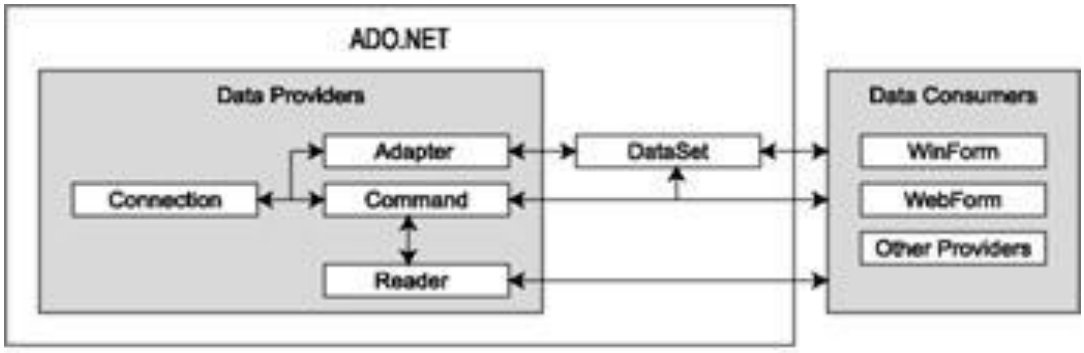
- SqlClient provides support for Microsoft SQL Server 7.0 and later. Use `System.Data.SqlClient`
- OracleClient provides support for Oracle 8.1.7 and later. Use `System.Data.OracleClient`
- OleDb (Object Linking & Embedding) provides support for connecting to OleDb sources like SQL server, Oracle, and Microsoft.ACE.OLEDB.12.0 (Microsoft Access). Use `System.Data.OleDb`
- ODBC (Open Database Connectivity) provides support for connection to ODBC source. Only used when you are not connecting to an OleDb source. Use `System.Data.ODBC`



ADO.NET Components

The following are the main ADO.NET abstractions:

- The ADO.NET **DataProviders** are components (namespaces containing classes) specifically designed for working with a particular database platform. There is a namespace for working with Access, SQL Server, Oracle, etc.
- The ADO.NET **DataSet** is a component used for working with data separate from the data source. It is an integral part of the disconnected data architecture. Data in the Dataset is independent of the database that was used to retrieve it
- The ADO.NET **DataReader** is an abstraction that is the main feature for working in the connected mode of DB processing. i.e. it is used to retrieves query results in a read-only, forward-only stream of data coming from a database

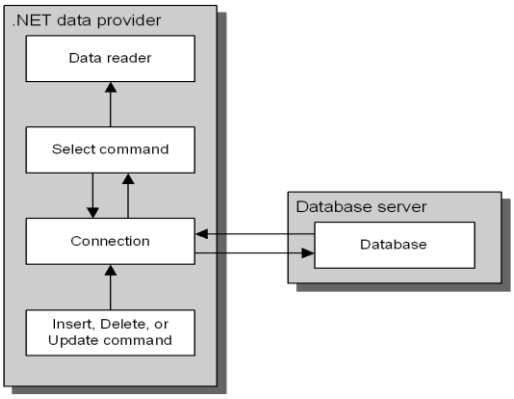


Connected and Disconnected Modes of Database Processing

ADO.NET contains two different approaches to work with data in a database: Connected Data Architecture and the Disconnected Data Architecture.

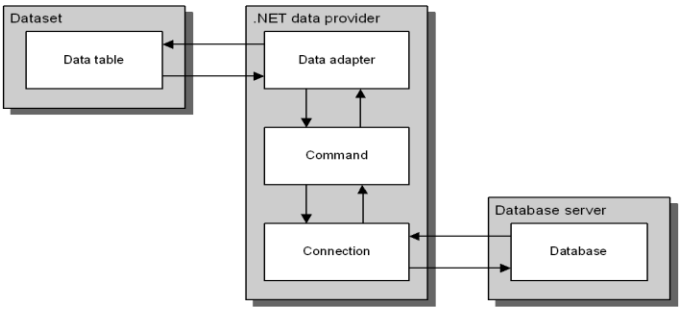
a) Connected Data Architecture:

Represents the objects that insist on having an open connection available for them to work and interact with the data source. ADO.NET provides classes to communicate directly with a data source. The following diagram depicts ADO.NET Components for the connected Approach:



b) Disconnected Data Architecture:

Represents the objects that open and close a connection to the data source as needed. Your application will work with data that was copied from the data source. Any changes made to the "copied" data in the Dataset will be later reconciled with the data source. The following diagram depicts ADO.NET Components for disconnected Approach:



The Primary Objects

- The *Connection Object* – Directs communication between your program and Data Source. Handles location and connection parameters for the data source.
- The *Command Object* – Takes an SQL statement you provide (as a string) and prepares it for transport through the Connection Object and subsequent processing in the specified DBMS.
- The *DataAdapter Object* – Enables communication between a DataSet and the rest of the Provider. Modifies SELECT, DELETE, INSERT, and UPDATE statements for use by related data source. It allows DataSet and DataTable to be filled from the data source and later reconciled with the data source.

The OleDbConnection Class

A connection is represented by the *System.Data.OleDb.OleDbConnection* class

Properties

i) The ***ConnectionString*** property contains a string that determines how ADO.NET will establish the database connection. One of the benefits of ADO.NET is that if a program is migrated to use a different database, all of the other code remains intact. Only the ConnectionString needs to be changed.

ii) The ***ConnectionTimeout*** specifies the number of seconds that can elapse to establish a connection.

iii) The ***DataSource*** property gets the location and file name of the database. This file name is embedded into the ConnectionString.
iv) The ***Provider*** property defines the database provider. This information is also embedded into the ConnectionString property.
v) The ***State*** property gets the state of the connection.

Methods

The Open and Close methods open and close the connection, respectively.

The OleDbCommand Class

- The **OleDbCommand** class stores SQL statements
- The **Connection** property stores a reference to an **OleDbConnection**
- The **CommandText** property stores an SQL statement and the **CommandType** should be set to Text

Processing Database Elements

All elements of a data base, tables (entities), rows (sets of values), columns (database fields), and individual cells (database values) are represented as abstractions in .NET i.e. every table, row, and column are represented as instances of the abstractions (classes) that are part of the ADO.NET support structure but the abstractions using DataReaders are different from those involved when using DataSets

Database Processing Sequence

Holds for both connected and disconnected architectures

i) Establish a connection to your data source using the Connection object.
ii) Create an SQL command statement either using a string or the Command object.
iii) Execute the Command object within the context of the connected database.
iv) Process the results of the commands by retrieving/storing the data when necessary.
- Use the DataReader to read the records.
- Use the DataAdapter and DataSet for storage of data retrieved from the database, and for use in your program.

v) Close the connection and release resources utilized by all the objects.

Example:

```
' Access Connection
Dim strConnection As String = "provider=Microsoft.Jet.OLEDB.4.0;" & _
                              "Data Source=c:\path\Inventory.mdb;"
Dim myConnection As New OleDbConnection(strConnection)

' Access 2007 - 2010 Connection
Dim strConnection As String = "provider=Microsoft.ACE.OLEDB.12.0;" & _
                    "Data Source=c:\path\Inventory.accdb;"
Dim myConnection As New OleDbConnection(strConnection)

' SQL Server Connection
Dim strConnection As String = "server= computing.scit.jkuat.edu;" & _
                        "database=Inventory; User id=xxx;" & "Password=xxx;"
Dim myConnection As New SqlConnection(strConnection)
```

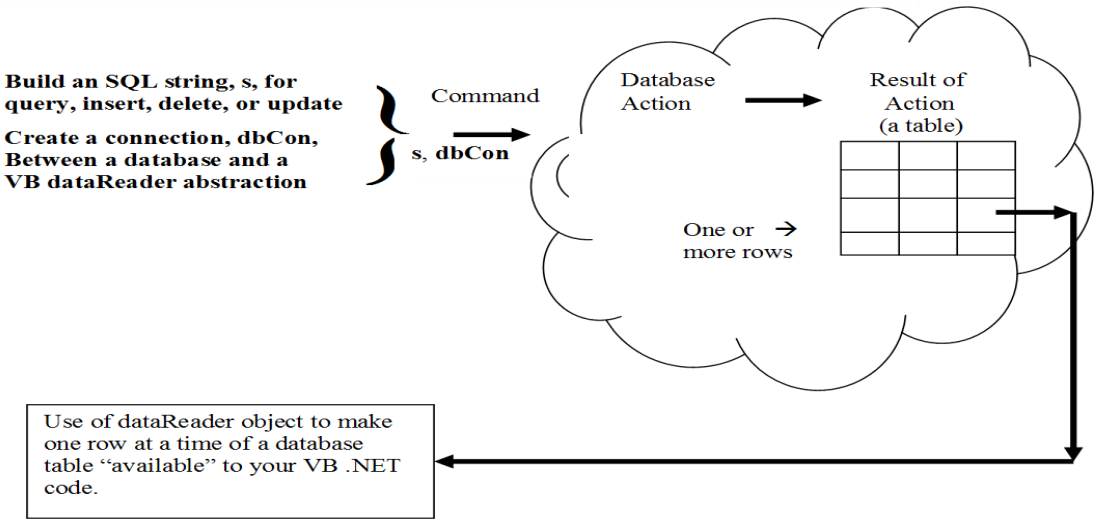
The connection string contains key-value pairs. The assignment operator (=) is used to separate the key and the value while a semi-colon (;) is used to separate each key-value pair.

DATABASE PROCESSING: THE CONNECTED MODEL (Using DataReaders)

- The DataReader is a component used for read-only and forward-only connection to a database.
- Results are returned as a query executes and stored in the network buffer on the client until you request them using the Read method of the DataReader.
- Using the DataReader can increase application performance both by retrieving data as soon as it is available, and (by default) storing only one row at a time in memory thus reducing system overhead.

- It is used to execute a query via a connection to a database and iterate through the data returned.

Using DataReaders in Processing Database Records



Examples

- Illustrations of various DB commands
- Create a connection, **dbCon**, between your (client) code and the database
- Build a database command (**SQL select, update, insert**, or **delete**) as a VB string s
- Send the string and connection information via a command to the provider

Example 1 – Insert (plus a simple select)

(Uses Insert and dataReader commands to effect data transmission)

- Direct insert of transaction into DB

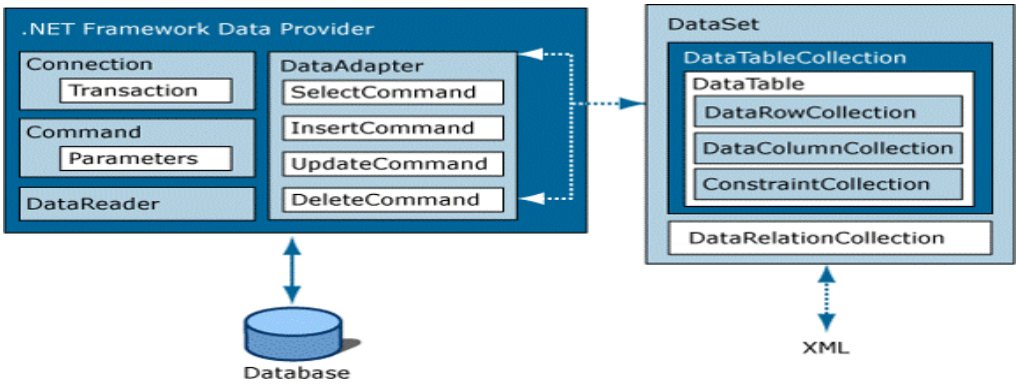
```
'Useful imports
Imports System.Data.OleDb
Imports System.Convert
Imports Microsoft.VisualBasic.ControlChars

...

' Write the Transaction to DB AND
' Get the MAX Transaction ID from DB to record transaction ID
Dim insertTransString As String = _
        "INSERT INTO tblTransaction (fldTransactionDate, " _
        "fldTransactionUser) VALUES (" & _
        thisDate & ", " & frmMain.thisUserID.ToString & ")"
' Create insert and read commands to be transmitted to DB
Dim insertTransCommand As New OleDbCommand(insertTransString, OleConn)
Dim readTransCommand As New OleDbCommand("SELECT MAX(fldTransactionID) "
        "FROM tblTransaction", OleConn)
' Create DataReader object
Dim thisReader As OleDbDataReader
Try
   OleConn.Open()    "OleConn is the Connection object
   insertTransCommand.ExecuteNonQuery()    ' Insert one transaction
   ' Next command executes SELECT to get Transaction Field ID
   thisReader = readTransCommand.ExecuteReader()
   ' Next - two methods needed on returned record from DataReader
   thisReader.Read()
   thisTransactionID = thisReader.GetInt32(0)   ' Gets first field read
   thisReader.Close()
Catch ex As Exception
   MessageBox.Show(ex.ToString)
Finally
   OleConn.Close()
End Try
...
```

Example 2 – Delete (remove) a record

```
Private Sub btnDelete_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnDelete.Click

    ' Create a valid SQL delete string to remove
    ' current record from the database
    Dim deleteString As String = _
        "DELETE FROM tblEmployees " & CrLf & _
        "WHERE tblEmployees.fldEmployeeID = " & (txtEmployeeID.Text)
    MsgBox(deleteString)

    ' Create a new OleDbCommand object to delete the data
    Dim deleteCommand As New _
         System.Data.OleDb.OleDbCommand(deleteString, OleConn)

    ' Use a try-catch to delete the record
    Try
        deleteCommand.ExecuteNonQuery()    'Directly changes the DB
        'MsgBox("Delete succeeded")
        ' Update result label
        lblResult.Text = "Employee deleted"
    Catch ex As Exception
        MessageBox.Show("Delete record error " & ex.ToString)
    End Try

    ' Reset the form
    reset()
End Sub
```

DATABASE PROCESSING: THE DISCONNECTED MODEL (Using DataAdapters, DataSets, and DataTables)

When working in the disconnected mode the data retrieved from a database and utilized by your program is stored in a DataSet (containing one or more DataTables). Data in the DataSet is independent of the database that was used to retrieve it and can contain the result of multiple queries. The connection to the database is typically closed after data is retrieved. The connection is opened again when it's needed.



DataAdapter

It is used to execute commands against a database and manages the transfer of information from the database to the DataSet. You the appropriate DataAdapter class.`OleDbDataAdapter, SqlDataAdapter, etc`

Example:

```
' Access Connection
Dim strConnection As String = "provider=Microsoft.Ace.OLEDB.12.0;" &
"Data Source=c:\path\Inventory.accdb;"

Dim myConnection As New OleDbConnection(strConnection)

Dim strSQL As String = "SELECT * FROM Product"

Dim myDataAdapter As New OleDbDataAdapter(strSQL, myConnection)
```

Note similarity with ExecuteQuery in connected mode

Using a Command with a DataAdapter

The command component is used to take an SQL command you provide, prepare it for transport through the connection object, and for processing in the DBMS. For our purposes, this is optional and can be easily replaced with a simple string.

Example:

```
' Access Connection
Dim strConnection As String = "provider=Microsoft.ACE.OLEDB.12.0;" & _
                              "Data Source=c:\path\Inventory.accdb;"

Dim myConnection As New OleDbConnection(strConnection)

Dim strSQL As String = "SELECT * FROM Product"

Dim myCommand As New OleDbCommand(strSQL, myConnection)

Dim myDataAdapter As New OleDbDataAdapter(myCommand)
```

Common properties of the SqlCommand class

Property	Description
Connection	The connection used to connect to the database.
CommandText	A SQL statement or the name of a stored procedure.
CommandType	A member of the CommandType enumeration that determines how the value in the CommandText property is interpreted.
Parameters	The collection of parameters for the command.

CommandType enumeration members

Member	Description
Text	The CommandText property contains a SQL statement. This is the default.
StoredProcedure	The CommandText property contains the name of a stored procedure.
TableDirect	The CommandText property contains the name of a table (OLE DB only).

Example:

```
Dim strSQL As String
Dim strConnection As String
Dim objDataSet As New DataSet()
Dim objAdapter As SqlDataAdapter
Dim objConnection As SqlConnection

strConnection = "server=computing.scit.jkuat.edu;Database=4376nn;" & _
"User id=4376nn;Password=yourpassword"

strSQL = "SELECT * FROM Product"
objConnection = New SqlConnection(strConnection)
objAdapter = New SqlDataAdapter(strSQL, objConnection)
objAdapter.Fill(objDataSet)
```

DataSet used to hold data retrieved from the database.

Connection object used to establish a connection.

DataAdapter object used to manage the flow of data from the data source to the DataSet.

Fills the DataSet object with data retrieved from the database.

DataSets

- Each DataSet contains one or more Tables i.e. Table[0], Table[1], Table[2] etc
- Each DataTable contain one or more rows i.e. Row[0], Row[1], Row[2] etc
- Each Row contains one or more fields i.e. These can be access by index [0]or by field name ["StudentName"]

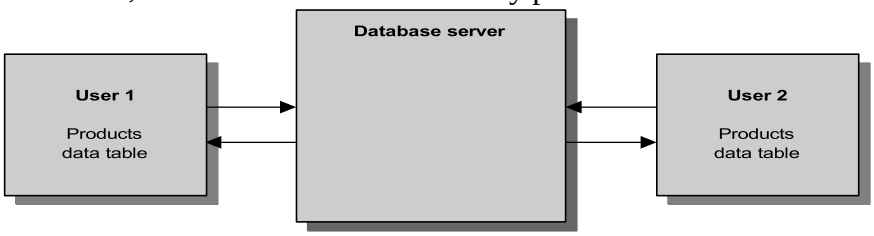
Comparisons of DataSets and DataReaders

- DataReaders retrieve data in read only form while Data in a DataSet may be modified in memory and updated in one step
- DataReaders allocate memory to one record (one table row) of data at a time i.e. efficient and little overhead. DataSets are less efficient. Space must be allocated for entire table involved.
- Only one record at a time can be processed. Random access through entire DataSet possible.
- Only one DataReader can be open at a time. Multiple DataSets can be open at once.
- Live connection to the database exists as long as the DataReader is open. Data connections maintained only long enough to transfer data between DataSet and database.

Advantages and Disadvantages of Disconnected Mode

i). The major advantage of the disconnected approach is that it improves system performance
- Uses less resources for maintaining connections when working with a data source.
- Work is done on local data: DataSets, DataTables, etc

ii). However, there is a disadvantage to this approach that occurs when two or more users try to update the same row of a table, which is called a concurrency problem.



Disconnected Mode Concurrency Problems

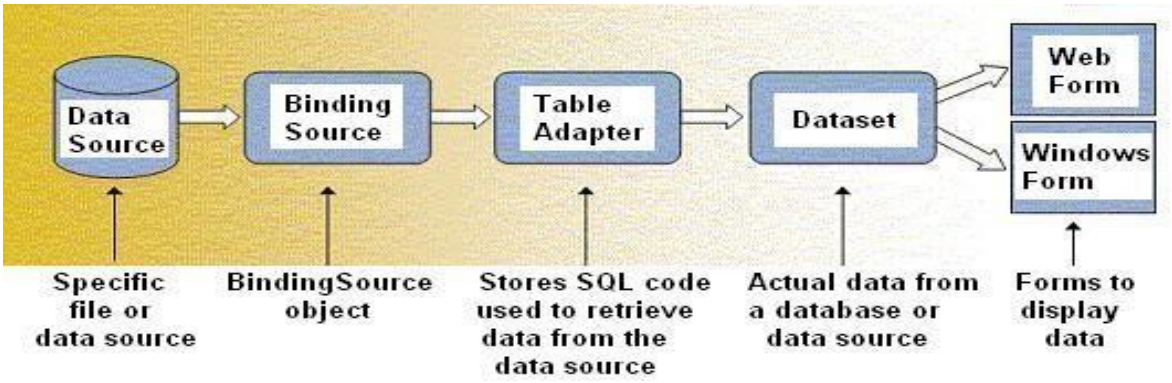
- When using the disconnected mode of DB access concurrency problems occur because once data is retrieved by User A from the database, the connection is dropped. Now User B has can access the DB
- There are two ways that ADO.NET can handle concurrency problems:
    1. Optimistic Concurrency – The program checks to see if there has been a change to the row since it was retrieved. If it has, the update or deletion will be refused and a concurrency exception will be thrown that your program must handle.
    2. "Last In Wins" – Since no checking is done, the row that was updated by the last user will overwrite changes made to the row by the previous user.
- Another way to avoid concurrency problems is to retrieve and update only one row at a time.

Wizards and Bound Data vs DataReaders and DataSets

- Wizard and data bound fields are useful for the rapid development of forms and displays based on content of a database but they are not useful for most transaction processing systems because of inflexibility
- Use of code to control database access is more powerful because the programmer can manipulate data before it is stored or after it is retrieved

Connecting to a Database or Data Source with VB

This figure below shows the steps in setting up a connection to a database or other type of data source.



- Configure a **binding source**. The binding source links to a **data source** – a data source is usually a specific database file, but can be another data source such as an array or text file.
- Configure a **table adapter**. A table adapter handles data retrieval and updating. The table adapter creates a dataset.
- Create a **dataset**. A dataset stores data rows that have been retrieved.
- Add controls to the form and set properties of the controls to **bind** the controls to the columns of a specific table in the dataset.
- VB will automatically write the code needed to fill a dataset.

Example

VB University Project – DataGridView Control Example

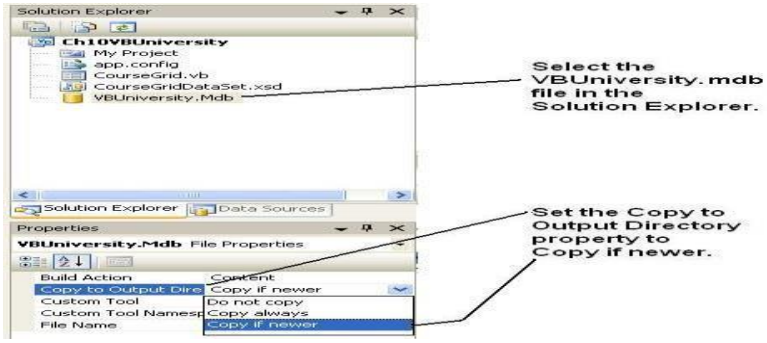
1. Add a **DataGridView** control from the tool box
2. Click the **smart arrow tag** in the upper right-corner of the DataGridView control – this displays the **DataGridView Tasks** window.
3. Modify the DataGridView Tasks window as follows:
   - **Dock** property – you can select either the value **Fill** in the **Properties** window or click the **Dock in parent container** link– this will cause the control to fill the form.
   - **Choose Data Source** – click the dropdown and select the **Add Project Data Source…** hyperlink – data sources can also be added with the **Data** menu.
4. **Data Source Configuration Wizard** – the wizard displays the **Choose a Data Source Type** window, select the **Database** as a data source type.
5. **Choose a Database Model** window – click **Dataset** option and click the Next button.
6. **Choose Your Data Connection** window – click the **New Connection** button. Your software may show an existing connection – you want a new connection for this project so ignore any existing connections.
7. Either **Choose Data Source** or an **Add Connection** window will open.
   - If a Choose Data Source window is displayed, select the **Microsoft Access Database File** option and click **Continue**.
8. If the Add Connection window is open or if you wish to change the type of connection to be created, then click the **Change** button. The default Data Source is **Microsoft SQL Server (SqlClient)**
9. For Microsoft Access:
   - In the **Add Connection** dialog box, click **Browse** – locate the **VBUniversity.mdb** file you earlier copied – select it and click **Open** in the browse window- you will see the path and database file name added to the **Add Connection** dialog box.
   - Click the **Test Connection** button – if the connection succeeds, you will see a popup window telling you that the **Test connection succeeded**. Click **OK** to close the popup and **OK** to close the **Add Connection** dialog box.

10. **Choose Your Data Connection** dialog box – you are returned to this window after selecting the database. Note that the name of the database file has been added to the window. This window shows an Access database file. Click **Next**.
11. Visual Studio now asks if you would like the database file (local data file) added to the project as shown below. Click **Yes**.
    - Clicking **Yes** causes the database file to copy to the project folder's root directory making the project portable– however, the file is copied every time you run the project so any changes you make in terms of adding new rows, modifying existing rows, or deleting rows will not be made permanently to the copy of the file without making additional modifications to the properties of the database file – we will make those changes after configuring the data source.
    - Clicking **No** causes the project to point (locate) the file based on the **ConnectionString** property setting in its original position – a copy of the database file is not made in the project.
12. **Save the Connection String to the Application Configuration File** – defaults to a selection of **Yes** – saving the connection string to a configuration file will enable you to change the string if necessary at a later point in time. Ensure the check box is **checked**, and then click **Next**.
13. **Choose Your Database Objects** – VB retrieves information from the database file about the tables and other database objects available for your use.
14. **Choose Your Database Objects** – you must specify the table(s) from which to retrieve the data to be displayed by the DataGridView control.
15. After the wizard closes, the DataGridView control now has column names from the **Course** table as column headings. Also the system component tray displays **BindingSource**, **TableAdapter**, and **DataSet** objects with names assigned by VB.

Changing the Database File's Properties

When the project runs, if you make changes to the data in the database, it is NOT saved with the current settings.

- **Copy to Output Directory** property setting – you must change this property of the **VBUniversity.mdb** or **VBUniversity.mdf** file as shown in the figure below.
- Change from **Copy always** to **Copy if newer** – when you make data row changes (inserts, edits, or deletes) during program execution, the changes will now be saved because the copy in the **\bin\Debug** folder will be the newest copy of the database file and the database file copy in the project root directory will not overwrite the database file copy in the **\bin\Debug** folder.



Coding the Course Form

VB automatically generates code for the form's **Load** event. The code is shown here.

- **Fill method** – the table adapter's **Fill** method executes the SQL statement that is stored in the table adapter. This fills the data set object with data from the **Course** table.
- VB generates the code automatically, but you can modify the code as necessary.

```
Private Sub CourseGrid_Load(sender As Object, ByVal e As EventArgs)
Handles MyBase.Load
    'TODO: This line of code loads data into the
    'CourseGridDataSet.Course' table. You can move, or remove it, as
```

```
        'needed.
        Me.CourseTableAdapter.Fill(Me.CourseGridDataSet.Course)
End Sub
```

Modify the **Load** event – add a **Try-Catch** block to handle situations where a network connection fails.

```
Private Sub CourseGrid_Load(sender As Object, ByVal e As EventArgs)
Handles MyBase.Load
        Try
            'Load Course table
            Me.CourseTableAdapter.Fill(Me.CourseGridDataSet.Course)
        Catch ex As Exception
            Dim MessageString As String = "Error: " &
             ControlChars.NewLine & ex.Message
            Dim TitleString As String = "Course Data Load Failed"
            MessageBox.Show(MessageString, TitleString,
            MessageBoxButtons.OK, MessageBoxIcon.Error)
        End Try
 End Sub
```

StudentGrid Form

1. Add a new form to the project and name it **StudentGrid.vb**.
2. Drag a **DataGridView** control to the form.
3. Click the **smart arrow tag** of the DataGridView control to display the DataGridView Tasks window.
   - **Dock** the DataGridView control to fill the entire form.
4. Choose a data source – use the **existing data connection** when the wizard displays the **Choose Your Data Connection** dialog box.
   - Work through the wizard. In the **Choose Your Database Objects** dialog box select the **Student** table.
   - Name the dataset **StudentGridDataSet**.
   - Click **Finish** – when the wizard closes, observe the new objects in the system component tray and the columns displayed in the DataGridView control.
5. Resize the form to make it large enough to display most of the data columns.
6. Open the coding window for the form and modify the **Load** event to add a **Try-Catch** block like the one shown here.

```
Private Sub StudentGrid_Load(sender As Object, e As EventArgs) Handles
MyBase.Load
        'Load the Student table
        Try
            Me.StudentTableAdapter.Fill(Me.StudentGridDataSet.Student)
        Catch ex As Exception
            Dim MessageString As String = "Error: " &
             ControlChars.NewLine & ex.Message
            Dim TitleString As String = "Student Data Load Failed"
            MessageBox.Show(MessageString, TitleString,
            MessageBoxButtons.OK, MessageBoxIcon.Error)
        End Try
    End Sub
```

Individual Data Fields – StudentDetails Form

Binding Data to Other Controls

- Another way to display data is with controls such as Label, TextBox and ComboBox controls – in this design approach, the controls are termed **bound controls**.
- The form can be designed through use of the **Data Sources** window – set the data source to display and drag the data table to the form.

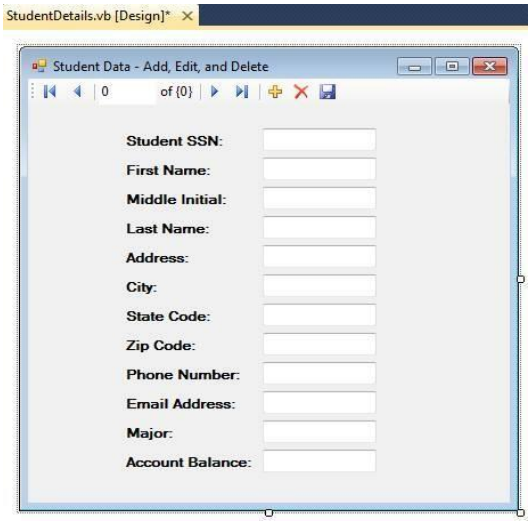
Add a New Form

1. **Project menu**, **Add Windows Form** option – click to add a new form named **StudentDetails.vb**.
2. **Text** property – set to **Student Data – Add, Edit, and Delete**.
3. Add **New Data Source** – add a new data source for this form. Using the **Data** menu, **Add New Data Source** menu option.
4. **Data Source Configuration** wizard:
   - Select the **Database** icon – click **Next**.
   - Choose a Database Model dialog box – click **Dataset** and then **Next**.
   - Choose Your Data Connection dialog box – click **Next** to use the existing connection.
   - Choose Your Database Objects dialog box – expand the **Tables** node as shown in the figure below – checkmark the **Student** table – name the dataset object **StudentDetailsDataSet** – click **Finish**.

Adding Bound Controls to the Form

You are ready to add bound controls to the form. You will use a drag/drop approach to build the form – this is really fast and easy to use.

1. **Data Sources** window –
   - Expand the **StudentDetailsDataSet**.
   - Single-click on the **Student** table name.
   - Select the drop-down arrow by the **Student** table and select **Details** (if the drop-down arrow does not display, ensure that the form view for design mode is displayed – not the coding view).
   - Note that the **icon** for a **Details** view is different than that for a **DataGridView**.
2. Point at the **Student** table name in the **Data Sources** window – **drag/drop** the table to the **StudentDetails** form to a position about an inch from the top and left margins of the form – the form will automatically have Label and TextBox controls generated and displayed for each column in the **Student** table as shown in this figure.

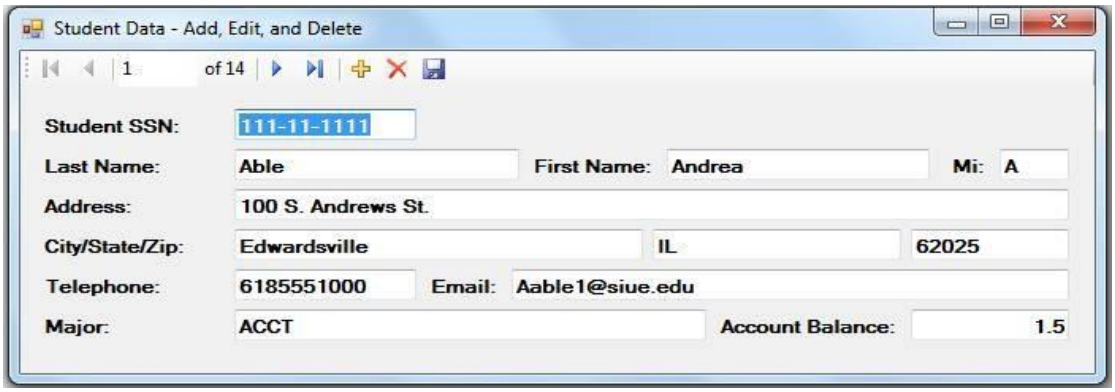


Note the following:

- The labels have the **Text** property setting generated automatically by VB – the column names are automatically divided into prompts appropriate for the form i.e a column named **StateCode** will generate a label prompt of **State Code:**.

- Note the new components that were automatically added to the system component tray: StudentDetailsDataSet, StudentBindingSource, StudentTableAdapter and TableAdapterManager StudentBindingNavigator – corresponds to the **StudentBindingNavigator** control that is automatically added across the top of the form. It has the following features:
    i) **Student record (row) counter**
    ii) **Move buttons** (Move first and Move previous are grayed out in the figure).
    iii) **Add**, **Delete**, and **Save button icons**.
- The form will run and execute, but the layout is not very "user-friendly."

3. Alter the layout by drag/drop the controls on the form as shown in the figure below.
    - Adjust the size of the controls as appropriate as shown here.
    - Reorder the name to display Last Name, then First Name, then Middle Initial.
    - Change the labels as shown on the figure for the Middle Initial (to Mi), City, State Code, and Zip Code (to City/State/Zip) and Phone Number (to Telephone).
    - **Tab order** – reset to reflect the new arrangement to tab from left-to-right, top-to-bottom.
    - **AccountBalanceTextBox** control – set **TextAlign** property to **Right**.



Individual Data Fields Coding
Form Load Event
The code generated by VB for the StudentGrid form's **Load** event needs to be modified by adding a Try-Catch block to catch exceptions as was done earlier for the Student form.

```
Private Sub StudentDetails_Load(sender As Object, ByVal e As EventArgs)
Handles MyBase.Load
    'Trap exceptions that occur during data load
    Try
        Me.StudentTableAdapter.Fill(Me.StudentDetailsDataSet.Student)
    Catch ex As Exception
        Dim MessageString As String = "Error: " & ControlChars.NewLine &
          ex.Message
        Dim TitleString As String = "Student Details Data Load Failed"
        MessageBox.Show(MessageString, TitleString,
        MessageBoxButtons.OK, MessageBoxIcon.Error)
    End Try
End Sub
```

Formatting String Data – Using a MaskedTextBox – Manually Setting Data Binding
You can use a **MaskedTextBox** control to format string output such as a telephone number, zip code, and social security number values.

Format Telephone Number
1. Delete the **Telephone TextBox** control and replace it with a **MaskedTextBox** control.
2. Name the control **TelephoneMaskedTextBox**.

3. Set the **Mask** property by opening the **Input Mask** window (click the **…** dialog box button) and select **Phone number** for a mask.

The new MaskedTextBox will not display any data because the control is not bound to the DataSource.

4. Select the **MaskedTextBox** control –
   - Expand the **DataBindings** property.
   - Select the **Text** property – Use the drop-down arrow for this property to select the **StudentBindingSource**, and then select the **PhoneNumber** column – this will bind the **Text** property so that it will display data from the **PhoneNumber** column of the **StudentBindingSource** object to the MaskedTextBox control
5. The same applies to **student SSN** and the **zip code** TextBox controls

Data Row Edit Operations

- The default configuration enables an application user to modify any data row, but does not provide a means to enforce saving changes when they are made.
- The **Save** button provided by default will save changes, but there is no way to ensure the application user clicks **Save** other than through the Form's **Closing** event.
- Using the Form's **Closing** event to save changes can result in an attempt to save multiple changes at the same time, but some of these may violate **database integrity rules**, such as the length of a data value or the type of data to be saved from a TextBox or other bound control.

Adding BindingNavigator Buttons

Add **EditToolStripButton** and **CancelToolStripButton** Controls

- Select the **BindingNavigator** control.
- Click the **Add ToolStripButton** drop-down on the control and select a **Button** to be added. This adds a **ToolStripButton**.
- Select the new **ToolStripButton** – set the **Name** property = **EditToolStripButton**.
- Right-click the **EditToolStripButton** button (the default image is a mountain with the sun shining) and either set the **Image** property to an appropriate image or switch the **DisplayStyle** property to **Text**.
- If you use Text instead of an image, set the **Text** property = **Edit**.
- Repeat these steps to add a second **Button** with **Name** property = **CancelToolStripButton**.

CancelToolStripButton Click Event

You must add a click event sub procedure for the new **CancelToolStripButton** control.

CancelEdit method – use this method of the **BindingSource** control to "throw away" any changes that may have been made to the current data row i.e. application users may wish to cancel when they accidentally click Edit or Add.

```
Private Sub CancelToolStripButton_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles CancelToolStripButton.Click
        'Cancel the operation
        StudentBindingSource.CancelEdit()
End Sub
```

Data Row Save Operations

This explains **Data Row Save Operations** with a **BindingNavigator** control. It is necessary to have a way to save changes made to a DataSet during both **Edit** and **Add** operations so that the changes are updated across the network to the database.

Coding the Save Button

- Visual Studio adds a **Save** Button to the **BindingNavigator** when the **TableAdapter** is generated when you drag the table onto the form.

- If you add a **BindingNavigator** control to a form, you will not see a **Save** Button and will need to add one.

The code generated by VB for the **Save** button of the binding navigator control is very simple and does not catch exceptions. The code is shown here.

```
Private Sub StudentBindingNavigatorSaveItem_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
StudentBindingNavigatorSaveItem.Click
        Me.Validate()
        Me.StudentBindingSource.EndEdit()
        Me.TableAdapterManager.UpdateAll(Me.StudentDetailsDataSet)
 End Sub
```

- The **Validate** method for the form confirms that any **Validating** event sub procedures have validated data on the form – on this project you will not use any **Validating** event sub procedures so this line of code has no purpose for your program – this line of code can be deleted or converted to a remark in case it is needed later.
- The **EndEdit** method of the **StudentBindingSource** object ends all edit operations. This applies to both updates of existing data rows and the addition of new data rows.
- The **UpdateAll** method of the **TableAdapterManager** control causes a reconnection to the database and updates any modifications of **StudentDetailsDataSet**.
- If an exception occurs, it will be caused by the **UpdateAll** method – possible exceptions include:
  - Trying to save a new row that duplicates an existing database row.
  - Trying to update a column value with an illegal value.

Using a Try-Catch Block

Modify the code by using a **Try-Catch** block to catch exceptions. The revised sub procedure is as follows:

```
Private Sub StudentBindingNavigatorSaveItem_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
StudentBindingNavigatorSaveItem.Click
        'Trap any exceptions during student table update
        Try
            'Me.Validate()
            Me.StudentBindingSource.EndEdit()
            Me.TableAdapterManager.UpdateAll(Me.StudentDetailsDataSet)
        Catch ex As Exception
            Dim MessageString As String = "Error: " &
            ControlChars.NewLine & ex.Message
            Dim TitleString As String = "Error During Save Operation"
            MessageBox.Show(MessageString, TitleString,
            MessageBoxButtons.OK, MessageBoxIcon.Error)
        End Try
    End Sub
```

The **UpdateAll** method is fairly complex and is more useful than the older **Update** method that was previously used to update datasets.

- Updates are performed on a **row-by-row basis** – this means that if multiple **DataSet** modifications (either Insert, Update, or Delete operations), then you would need to control the order of the updates.
- Using the **TableAdapterManager** ensures that updates involving multiple tables are processed in the correct order where the order of the updates is important.
- The **Structured Query Language** commands to execute the required **INSERT**, **UPDATE**, or **DELETE** statements on the database are generated automatically by Visual Studio and the **UpdateAll** method automatically selects the correct command to pass to the database.

- When an **UpdateAll** method fires, the **TableAdapterManager** examines each data row's **RowState** property in order to execute the required **INSERT**, **UPDATE**, or **DELETE** statements.
- After a successful update, changes to the data row are accepted within the **DataSet** – this makes it unnecessary to call the **AcceptChanges** method of the **DataSet** object.

Test the project.

- Run the project, select a data row to edit, and click the **Edit** Button.
- Make a change to the data (such as the student's Major field of study) and click the **Save** button.
- Quit the application, then start it again – you should find that the data row was saved with the new data value.

Data Row Delete Operations

This part explains **Data Row Delete Operations** with a **BindingNavigator** control. This operation does not use the **Save** or **Cancel** Buttons – rather, the code for the **Delete** button click event updates the database.

Changing BindingNavigatorDeleteItem Button Behavior

You should change the way the **Delete** Button on the **BindingNavigator** works in order to ask for application user confirmation a deletion before deleting a data row.

- When the **Button's Click** event executes, the **BindingNavigator** has already called the **RemoveCurrent** method for the **BindingSource** object and there is no simple way to cancel the action.
- The BindingNavigator has a **DeleteItem** property – when the **ToolStripItem** (**Delete Button**) associated with the property is clicked, the **RemoveCurrent** method is called.
- To change the behavior, clear the **DeleteItem** property to prevent implicitly calling the **BindingSource** object's **RemoveCurrent** method.

1. **StudentBindingNavigator** – select this component.
2. **Properties** window – set the **DeleteItem** property of the **StudentBindingNavigator** = (**none)** at the top of the list of enumerated values.

BindingNavigatorDeleteItem Click Event

Code a sub procedure to handle removal of data rows.

- **Delete Button** of the **BindingNavigator** control – double-click to create a **Click** event sub procedure and add the code shown below

```
Private Sub BindingNavigatorDeleteItem_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
BindingNavigatorDeleteItem.Click
        'Store the current DataSet position in case the deletion fails
        Dim RowNumberInteger As Integer = StudentBindingSource.Position
        Try
            Dim ResponseDialogResult As DialogResult =
            MessageBox.Show("Confirm to delete the student record.", _
            "Delete Y/N?", MessageBoxButtons.YesNo,
             MessageBoxIcon.Question, MessageBoxDefaultButton.Button2)
             If ResponseDialogResult = Windows.Forms.DialogResult.Yes Then
                'Delete the row by removing the current record,
                'ending the edit, and calling the Update method
                StudentBindingSource.RemoveCurrent()
                StudentBindingSource.EndEdit()
                TableAdapterManager.UpdateAll(StudentDetailsDataSet)
            End If
         Catch exOleDb As OleDb.OleDbException
             'Restore the deleted row with the RejectChanges method
            StudentDetailsDataSet.RejectChanges()
```

```
            'Reposition to the row that was deleted
            StudentBindingSource.Position = RowNumberInteger
            'Display appropriate error message
            MessageBox.Show("This student cannot be deleted - the student
            is enrolled in courses." & ControlChars.NewLine &  _
            exOleDb.Message, "Delete Operation Error", _
             MessageBoxButtons.OK, MessageBoxIcon.Error)
        Catch ex As Exception
            'Some other exception was triggered
            MessageBox.Show("Unexpected error in delete operation: " &
            ControlChars.NewLine & ex.Message, "Delete Operation Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error)
    End Try
End Sub
```

- Assume that the delete operation may **fail** – it will be necessary to **restore** the dataset and redisplay the deleted record. The first line of code saves the record number of the data row to be deleted.
- If the application user responds to delete the row, the value of **ResponseDialogResult** is checked. If it is "**Yes**", then:
  - The **RemoveCurrent** method removes the row.
  - The **EndEdit** method ends the edit.
  - The **UpdateAll** method of the **TableAdapterManager** updates the **StudentDetailsDataSet** object (This can also be coded using the Update method of the **StudentTableAdapter**:
    StudentTableAdapter.Update(StudentDetailsDataSet.Student)
- If the deletion fails, the line of code generating the exception will usually be the line of code with the **UpdateAll** method. This triggers an **OleDbException** that is caught by the first of two **Catch** blocks. The **OleDbException** is raised when a deletion fails due to referential integrity constraints:
  - A message box displays an appropriate message.
  - The **DataSet's RejectChanges** method is used to reject the deletion in the DataSet – essentially this **undeletes** the deleted row that the database rejected.
  - The **BindingSource's Position** property is reset to the record number of the data row for which deletion failed.