ADO.NET in Visual Basic

Source code

- Download the source code of the tutorial from the "Esercitazioni" page of the course web page
- http://www.unife.it/ing/lm.infoauto/sistemi-informativi/allegati/ADO_Tutorial.zip
- Uncompress
- Select "ADO Tutorial.sln"
- Right click-> open with -> Visual Studio 2010

Preliminaries

- First you have to import the namespace containing the ADO.NET classes
- Namespace with ADO.NET classes

Imports System.Data

- Namespace with ADO.NET classes specific for SQL Server (.NET Framework Data Provider for SQL Server)

Imports System.Data.SqlClient

Connection

- In ADO.NET you use a **Connection** object to connect to a specific data source by supplying necessary authentication information in a connection string. The **Connection** object you use depends on the type of data source.
- To connect to Microsoft SQL Server 7.0 or later, use the SqlConnection object of the .NET Framework Data Provider for SQL Server

Connection

```
Dim connection As New SqlConnection( _
"Data Source=10.17.2.91; User ID =si;” & _
    “Password=sistemi;" & _
"Initial Catalog=AdventureWorks")
connection.Open()
```

Connection String Keywords

- “Data source” or “server”= name of database server, optionally followed by \instance_name
- “User ID” and “Password”: SQL Server authentication, for example:
  - “User ID=myuser;Password=mypassword;Initial Catalog=AdventureWorks;Server=MySqlServer“
  - Username must be an SQL Server login
- “Integrated Security=true”: Windows authentication is used, no username and password must be specified
- “Initial Catalog” or “Database”= the database to connect to

Commands

- After establishing a connection to a data source, you can execute commands and get results from the data source using a **Command** object.
- You can create a command using the **Command** constructor, which takes as arguments (both optional)
  - an SQL statement to execute at the data source,
  - a **Connection** object.
- You can also create a command for a particular connection using the **CreateCommand** method of the **Connection** object.

Creating a Command

```
Dim cmd As New SqlCommand( _
    “SELECT * FROM Person.Contact”, _
    connection)
```

Execute Methods

- The **Command** object exposes several **Execute** methods that you can use to perform the intended action.
  - When returning results as a stream of data, use **ExecuteReader** to return a **DataReader** object.
  - Use **ExecuteScalar** to return a singleton value.
  - Use **ExecuteNonQuery** to execute commands that do not return rows

ExecuteReader

```
Dim reader As SqlDataReader = cmd.ExecuteReader()
Try
    While reader.Read()
        Console.WriteLine(String.Format("{0}, {1}", _
        reader(0), reader(1)))
    End While
Finally ' Always call Close when done reading.
    reader.Close()
End Try
```

DataReader

- You can use the ADO.NET **DataReader** to retrieve a read-only, forward-only stream of data from a database.
- Results are returned as the query executes, and are stored in the network buffer on the client until you request them using the **Read** method of the **DataReader**.

DataReader

- You use the **Read** method of the **DataReader** object to obtain a row from the results of the query.
- You can access each column of the returned row by passing the name or ordinal reference of the column to the **DataReader**.

ExecuteReader

```
Dim reader As SqlDataReader = cmd.ExecuteReader()
Try
   While reader.Read()
         Console.WriteLine(String.Format("{0}, {1}", _
          reader("ContactID"),reader("NameStyle")))
   End While
Finally ' Always call Close when done reading.
   reader.Close()
End Try
```

ExecuteReader

- For best performance, the **DataReader** provides a series of methods that allow you to access column values in their native data types (**GetDateTime**, **GetDouble**, **GetGuid**, **GetInt32**, and so on).
- They take only the column number, not the column name

DataReader

```
Do While reader.Read()
    Console.WriteLine(vbTab & "{0}" & vbTab & "{1}", _
    reader.GetInt32(0), reader.GetBoolean(1))
Loop
reader.Close()
```

DataReader

- The **DataReader** is a good choice when retrieving large amounts of data because the data is not cached in memory.
- You should always call the **Close** method when you have finished using the **DataReader** object.
- Note that while a **DataReader** is open, the **Connection** is in use exclusively by that **DataReader**. You cannot execute any commands for the **Connection**, including creating another **DataReader**, until the original **DataReader** is closed.

Multiple Result Sets

- If multiple result sets are returned, the **DataReader** provides the **NextResult** method to iterate through the result sets in order.

Multiple Result Sets

```
Dim command As SqlCommand = New SqlCommand( _
"SELECT CurrencyCode, Name FROM Sales.Currency;" & _
"SELECT DepartmentID, Name FROM HumanResources.Department", _
    connection)
Dim reader As SqlDataReader = command.ExecuteReader()
Dim nextResult As Boolean = True
Do Until Not nextResult
    Console.WriteLine(vbTab & reader.GetName(0) & vbTab & _
            reader.GetName(1))
        Do While reader.Read()
            Console.WriteLine(vbTab & reader(0) & vbTab & _
            reader.GetString(1))
        Loop
        nextResult = reader.NextResult()
Loop
reader.Close()
```

Returning a Single Value

- You may need to return database information that is simply a single value rather than in the form of a table or data stream.
- For example, you may want to return the result of an aggregate function such as COUNT(*), SUM(Price), or AVG(Quantity).
- The **Command** object provides the capability to return single values using the **ExecuteScalar** method.
- The **ExecuteScalar** method returns as a scalar value the value of the first column of the first row of the result set

Returning a Single Value

```
' Assumes that connection is a valid SqlConnection
    object.
Dim ordersCMD As SqlCommand = New _
    SqlCommand( _
  "SELECT COUNT(*) FROM Sales.Store", connection)
Dim count As Int32 = CInt(ordersCMD.ExecuteScalar())
Console.WriteLine("Number of stores={0}", count)
```

Modifying Data

- You can execute stored procedures or data definition language statements (for example, CREATE TABLE and ALTER COLUMN)
- You can execute INSERT, UPDATE and DELETE statements
- These commands do not return rows as a query would, so the **Command** object provides an **ExecuteNonQuery** to process them.

Creating a Table

```
Dim connection As New SqlConnection( _
        "Data Source=10.17.2.91;User ID=si;” & _
        “Password=sistemi;" & _
        "Initial Catalog=prova").
connection.Open()
Dim queryString As String = "CREATE TABLE “ & _
   “IMPIEGATI_MAT " & _
 "(ID INT PRIMARY KEY, NOME VARCHAR(20), COGNOME
   VARCHAR(20), CITTA VARCHAR(50), ETA INT)"
     Dim command As SqlCommand = New
   SqlCommand(queryString, connection)
     command.ExecuteNonQuery()
```

Modifying Data

```
Dim connection As New SqlConnection( _
        "Data Source=10.17.2.91;User ID=si;” & _
        “Password=sistemi;" & _
        "Initial Catalog=prova")
connection.Open()
Dim queryString As String = "INSERT INTO " & _
 "IMPIEGATI_MAT " & _
 "Values(1,'Mario', 'Rossi', 'Ferrara', 30)"
Dim command As SqlCommand = New _
   SqlCommand(queryString, connection)
Dim recordsAffected As Int32 = command.ExecuteNonQuery()
Console.WriteLine("{0} records affected", _
   recordsAffected)
```

Using Parameters

- The ? syntax for parameters can not be used
- Parameters must have a name
- Each SqlCommand has a list of parameters associated to it
- They must be explicitly added to the parameters list
- Then their value can be set

Using Parameters

```
Dim connection As New SqlConnection( _
"Data Source=10.17.2.91; User ID =si;” & _
“Password=sistemi; " & _
"Initial Catalog=AdventureWorks")
connection.Open()
Dim cmd As New SqlCommand("SELECT * FROM “& _
   “Person.Contact C where C.ContactID=@ID", _
         connection)
 connection.Open()
 cmd.Parameters.Add("@ID", SqlDbType.Int)
 cmd.Parameters("@ID").Value = 1
```

Using Parameters

```
Dim reader As SqlDataReader = cmd.ExecuteReader()
Try
  While reader.Read()
      Console.WriteLine(String.Format("{0}, {1}",
  reader(0), reader(1)))
  End While
Finally ' Always call Close when done reading.
   reader.Close()
End Try
```

Reuse of SqlCommand

- You can reset the **CommandText** property and reuse the **SqlCommand** object.
- However, you must close the SqlDataReader before you can execute a new or previous command.

Reuse of SqlCommand

```
Dim connection As New SqlConnection( _
        "Data Source=10.17.2.91;User ID=si;Password=sistemi;" & _
       "Initial Catalog=prova")
   connection.Open()
   Dim queryString As String = "INSERT INTO " & _
"IMPIEGATI_MAT " & _
"Values(2,'Andrea', 'Bianchi', 'Rovigo', 31)"
   Dim command As SqlCommand = New SqlCommand(queryString,
  connection)
   Dim recordsAffected As Int32 = command.ExecuteNonQuery()
   Console.WriteLine("{0} records affected", recordsAffected)
   command.CommandText = " INSERT INTO " & _
"IMPIEGATI_MAT " & _
"Values(3,'Giovanni', 'Verdi', 'Bologna', 40)"
   recordsAffected = command.ExecuteNonQuery()
   Console.WriteLine("{0} records affected", recordsAffected)
```

DataSet

- The ADO.NET DataSet is a memory-resident representation of data that provides a consistent relational programming model regardless of the source of the data it contains.
- A **DataSet** represents a complete set of data including the tables that contain, order, and constrain the data, as well as the relationships between the tables.
- A **DataSet** can be populated with tables of data from an existing relational data source using a **DataAdapter**

DataSet

- A DataSet object is a collection of DataTable objects
- A DataTable object stores a table of data
- A DataSet object contains also information on the relations among the tables and on the constraints.

Populating a DataSet

- By means of a **DataAdapter**
- To create a **DataAdapter**, pass a string containing an SQL command and an open connection to the constructor.
- Alternatively, pass a **Command** object. This will be stored in the **SelectCommand** property
- The **Fill** method of the **DataAdapter** is used to populate a **DataSet** with the results of the **SelectCommand** of the **DataAdapter**. **Fill** takes as its arguments a **DataSet** to be populated, and a **DataTable** object, or the name of the **DataTable** to be filled with the rows returned from the **SelectCommand**.

Populating a DataSet

```
Dim connection As New SqlConnection( _
        "Data Source=10.17.2.91;User ID=si;Password=sistemi;" & _
         "Initial Catalog=prova")
 connection.Open()
 Dim queryString As String = _
 "SELECT* FROM " & _
 "IMPIEGATI_MAT"
      Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
        queryString, connection)
      Dim impiegati As DataSet = New DataSet
      adapter.Fill(impiegati, "IMPIEGATI_MAT")
```

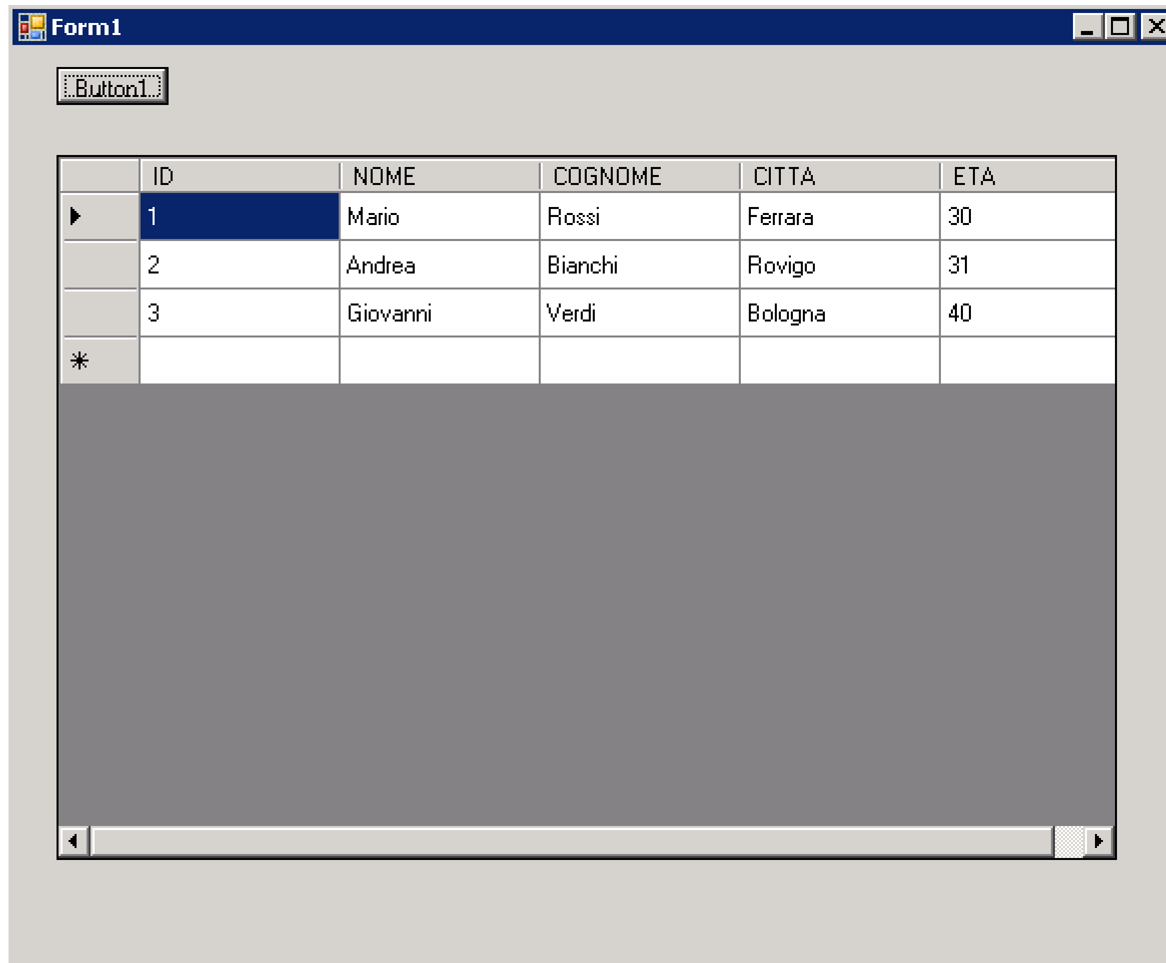
Showing Data

- To show data to the user, use a **DataGridView**
- If grdDemo is an object of type DataGridView

```
grdDemo.DataSource = impiegati
grdDemo.DataMember = "IMPIEGATI_MAT"
```

The first statement select the DataSource of the DataGridView, the second selects the table to show

DataGridView



Editing Data in a DataSet

- You have to choose a DataTable
- You can access the content of a DataTable by using the **Rows** collection of the **DataTable**.
- ds.Tables(“TableName”) returns the **DataTable** object with name “TableName” from the DataSet ds
- ds.Tables(“TableName”).Rows(0) returns the row number 0 from “TableName”
- ds.Tables(“TableName”).Rows(0)(ColumnNumber)
- ds.Tables(“TableName”).Rows(0)(ColumnName)
- Return the value of the column with ColumnNumber or ColumnName

Editing Data in a Dataset

```
Dim row As DataRow = _
   impiegati.Tables(“IMPIEGATI_MAT").Rows(0)
row(“Nome") = “Maria"
```

Adding a Row

- You can add new rows of data to a DataTable.
- To add a new row, declare a new variable as type DataRow.
- A new **DataRow** object is returned when you call the NewRow method of DataTable.
- The **DataTable** then creates the **DataRow** object based on the structure of the table
- You then can manipulate the newly added row using the column index or the column name
- After data is inserted into the new row, the **Add** method is used to add the row to the DataRowCollection.

Adding a Row

```
Dim imp As DataTable = _
impiegati.Tables("IMPIEGATI_MAT")
Dim workRow As DataRow = imp.NewRow()
workRow(“ID”)=4
workRow("NOME") = "Stefano"
workRow(2) = "Zucchi"
workRow(3) = "Roma"
workRow("ETA") = 25
imp.Rows.Add(workRow)
```

Deleting a Row

- Use the Delete method of the **DataRow** object.
- The **Delete** method marks the row for deletion.

Deleting a Row

```
Dim imp As DataTable = _
    impiegati.Tables("IMPIEGATI_MAT")
    For Each row As DataRow In imp.Rows
        If row(2) = "Verdi" Then
            row.Delete()
        End If
    Next
```

Changes to the Database

- Updating the DataSet does not update the database from which the data was taken to populate it

RowState

- Each DataRow object has a RowState property that you can examine to determine the current state of the row.
- Moreover, a row can have various version
- For example, after you have made a modification to a column in a row, the row will have a row state of **Modified**, and two row versions: **Current**, which contains the current row values, and **Original**, which contains the row values before the column was modified.

RowState

- Main row states:
  - **Unchanged:** No changes have been made since it was created by DataAdapter.Fill.
  - **Added**: The row has been added to the table
  - **Modified**: Some element of the row has been changed
  - **Deleted**: The row has been deleted from a table

Updating the Data Source

- The **Update** method of the DataAdapter is called to resolve changes from a DataSet back to the data source.
- The **Update** method, like the **Fill** method, takes as arguments an instance of a **DataSet**, and an optional **DataTable** object or **DataTable** name. The **DataSet** instance is the **DataSet** that contains the changes that have been made, and the **DataTable** identifies the table from which to retrieve the changes.

Updating the Data Source

- When you call the **Update** method, the **DataAdapter** analyzes the changes that have been made and executes the appropriate command (INSERT, UPDATE, or DELETE).
- When the **DataAdapter** encounters a change to a DataRow, it uses the **InsertCommand**, **UpdateCommand**, or **DeleteCommand** to process the change.
- This allows you to maximize the performance of the ADO.NET application by specifying command syntax at design-time and, where possible, through the use of stored procedures.
- You must explicitly set the commands before calling **Update**.

Command Generation

- If **Update** is called and the appropriate command does not exist for a particular update (for example, no **DeleteCommand** for deleted rows), an exception is thrown
- If your **DataTable** maps to or is generated from a single database table, you can take advantage of the **SqlCommandBuilder** object to automatically generate the **DeleteCommand**, **InsertCommand**, and **UpdateCommand** of the **DataAdapter**.
- The table schema retrieved by the **SelectCommand** property determines the syntax of the automatically generated INSERT, UPDATE, and DELETE statements

Command Generation

- The **SqlCommandBuilder** must execute the **SelectCommand** in order to return the metadata necessary to construct the INSERT, UPDATE, and DELETE SQL commands.
- As a result, an extra trip to the data source is necessary, which can hinder performance. To achieve optimal performance, specify your commands explicitly rather than using the **SqlCommandBuilder**
- The **SelectCommand** must also return at least one primary key or unique column. If none are present, an **InvalidOperation** exception is generated, and the commands are not generated.

Command Generation

```
Dim connection As New SqlConnection( _
          "Data Source=10.17.2.91;User ID=si;” & _
          “Password=sistemi;" & _
          "Initial Catalog=prova")
      connection.Open()
Dim queryString As String = _
  "SELECT * FROM " & _
  "IMPIEGATI_MAT"
Dim adapter As SqlDataAdapter = New
    SqlDataAdapter( _
         queryString, connection)
```

Command Generation

```
Dim builder As SqlCommandBuilder = New
  SqlCommandBuilder(adapter)
    builder.QuotePrefix = "["
    builder.QuoteSuffix = "]"
Dim impiegati As DataSet = New DataSet
    adapter.Fill(impiegati, "IMPIEGATI_MAT")
    grdDemo.DataSource = impiegati
    grdDemo.DataMember = "IMPIEGATI_MAT"
```

Update Command

- Updates rows at the data source for all rows in the table with a **RowState** of Modified. Updates the values of all columns except for columns that are not updateable, such as identities or expressions. Updates all rows where the column values at the data source match the primary key column values of the row, and where the remaining columns at the data source match the original values of the row.

Update

```
Dim row As DataRow = _
        impiegati.Tables("IMPIEGATI_MAT").Rows(0)
row("Nome") = "Marianna"
' Without the SqlCommandBuilder, this line would fail.
    adapter.Update(impiegati, "IMPIEGATI_MAT")
```

Insert Command

- Inserts a row at the data source for all rows in the table with a **RowState** of Added. Inserts values for all columns that are updateable (but not columns such as identities, expressions, or timestamps).

Insert

```
Dim imp As DataTable = _
    impiegati.Tables("IMPIEGATI_MAT")
    Dim workRow As DataRow = imp.NewRow()
    workRow("ID") = 5
    workRow("Nome") = "Andrea"
    workRow(2) = "Biagi"
    imp.Rows.Add(workRow)
    ' Without the SqlCommandBuilder, this line would
  fail.
    adapter.Update(impiegati, "IMPIEGATI_MAT")
    connection.Close()
```

Delete Command

- Deletes rows at the data source for all rows in the table with a **RowState** of Deleted. Deletes all rows where the column values match the primary key column values of the row, and where the remaining columns at the data source match the original values of the row.

Delete

```
Dim imp As DataTable =
    impiegati.Tables("IMPIEGATI_MAT")
        For Each row As DataRow In imp.Rows
            If row(1) = "Marianna" Then
                row.Delete()
            End If
        Next
        ' Without the SqlCommandBuilder, this line would
    fail.
 adapter.Update(impiegati, "IMPIEGATI_MAT")
```

Optimistic Concurrency Control

- The logic for generating commands automatically for UPDATE and DELETE statements is based on *optimistic concurrency*--that is, records are not locked for editing and can be modified by other users or processes at any time.
- Because a record could have been modified after it was returned from the SELECT statement, but before the UPDATE or DELETE statement is issued, the automatically generated UPDATE or DELETE statement contains a WHERE clause, specifying that a row is only updated if it contains all original values and has not been deleted from the data source. This is done to avoid new data being overwritten.
- Where an automatically generated update attempts to update a row that has been deleted or that does not contain the original values found in the DataSet, the command does not affect any records and a DBConcurrencyException is thrown.

Manually Setting the Update Commands

- To specify a different concurrency control, the update commands can be manually set

Transactions

- **To perform a transaction**
  1. Call the BeginTransaction method of the SqlConnection object to mark the start of the transaction. The **BeginTransaction** method returns a reference to a SqlTransaction object.
  2. Assign the **SqlTransaction** object to the Transaction property of the **SqlCommand** to be executed. If a command is executed on a connection with an active transaction, and the **SqlTransaction** object has not been assigned to the **Transaction** property of the **Command** object, an exception is thrown.

Transactions

3. Execute the required commands.
4. Call the Commit method of the SqlTransaction object to complete the transaction, or call the Rollback method to abort the transaction. If the connection is closed or disposed before either the **Commit** or **Rollback** methods have been executed, the transaction is rolled back.

Transactions

```
Dim connection As New SqlConnection( _
     "Data Source=10.17.2.91;User ID=si;Password=sistemi;" & _
       "Initial Catalog=prova")
    connection.Open()
    ' Start a local transaction.
    Dim sqlTran As SqlTransaction = _
  connection.BeginTransaction()
    ' Enlist the command in the current transaction.
    Dim command As SqlCommand = _
  connection.CreateCommand()
    command.Transaction = sqlTran
```

Transactions

```
 Try
        command.CommandText = _
          "INSERT INTO IMPIEGATI_MAT(ID,NOME) “ & _
    ”VALUES(6,'Pietro')"
        command.ExecuteNonQuery()
command.CommandText = _
      "INSERT INTO IMPIEGATI_MAT(ID,NOME) “ & _
    VALUES(7,'Anna')"
        command.ExecuteNonQuery()
        sqlTran.Commit()
        Console.WriteLine("Both records were written to
    database.")
```

Transactions

```
Catch ex As Exception
        Console.WriteLine(ex.Message)
        Console.WriteLine("Neither record was  " & _
  "written to database.")
        sqlTran.Rollback()
End Try
```