

3

Unit 3 - ADO.NET, Working with XML

▼ Syllabus

ADO.NET: Data Provider Model, Direct Data Access - Creating a Connection, Select Command, DataReader, Disconnected Data Access

Data Binding: Introduction, Single-Value Data Binding, Repeated-Value Data Binding

Data Source Controls - SqlDataSource Data Controls: GridView, Details View, Form View

Working with XML: XML Classes - XMLTextWriter, XMLTextReader Caching: When to Use Caching, Output Caching, Data Caching

ADO.NET



ADO.NET is the technology that .NET applications use to interact with a database.

- ADO.NET is a technology designed to let an ASP.NET program (or any other .NET program, for that matter) access data.

Understanding the Data Provider Model

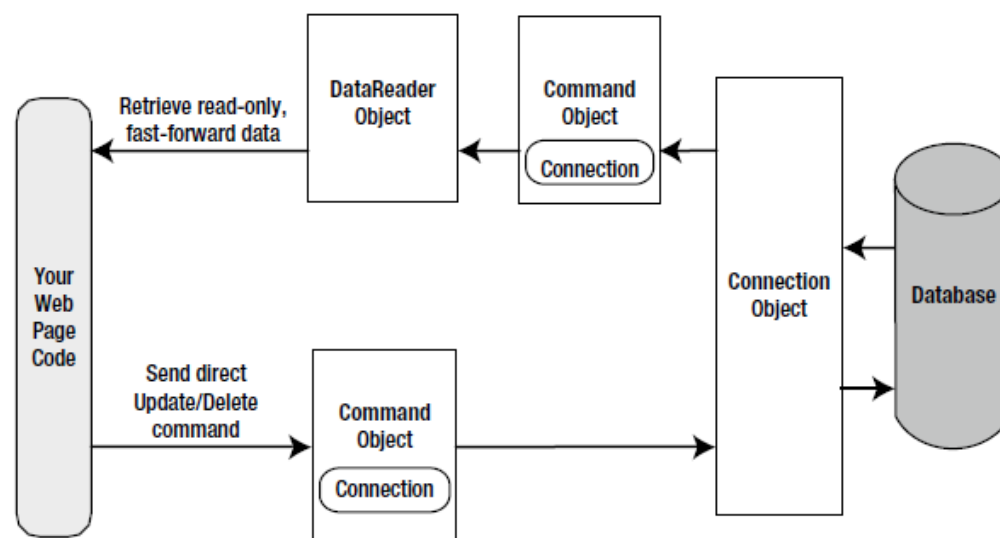
- ADO.NET relies on the functionality in a small set of core classes.
- You can divide these classes into two groups:
 - used to contain and manage data
 - used to connect to a specific data source
- `System.Data.SqlClient` : Contains the classes you use to connect to a Microsoft SQL Server database and execute commands.
- `System.Data.SqlTypes` : Contains structures for SQL Server-specific data types. You can use these types to work with SQL Server data types without needing to convert them into the standard .NET equivalents.
- `System.Data` : Contains fundamental classes with the core ADO.NET functionality. These allow you to manipulate structured relational data.

Using Direct Data Access (Connected)



The most straightforward way to interact with a database is to use direct data access.

- When you query data with direct data access, you don't keep a copy of the information in memory.
- You work with it for a brief period of time while the database connection is open, and then close the connection as soon as possible.
- Well suited to ASP.NET web pages, which don't need to keep a copy of their data in memory for long periods of time.
- **Steps to query information with direct data access**
 1. Create `Connection`, `Command`, and `DataReader` objects.
 2. Use the `DataReader` to retrieve information from the database, and display it in a control on a web form.
 3. Close your connection.
 4. Send the page to the user. (At this point, all the ADO.NET objects have been destroyed)
- **Steps to add or update information with direct data access**
 1. Create new `Connection` and `Command` objects.
 2. Execute the `Command` (with the SQL statement).



1. Creating a Connection

- Before you can retrieve or update data, you need to make a connection to the data source.
- Generally, connections are limited to some fixed number, therefore you should try to hold a connection open for as short a time as possible.



Write your database code inside a try/catch error-handling structure so you can respond if an error does occur, and make sure you close the connection even if you can't perform all your work.

- When creating a `Connection` object, you need to specify a value for its `ConnectionString` property.
- `ConnectionString` defines all the information the computer needs to find the data source, log in, and choose an initial database.

```

SqlConnection myConnection = new SqlConnection();
myConnection.ConnectionString = "Data Source=MUM0212CPU0558\\SQLEXPRESS;" +
                               "Initial Catalog=students; " +
                               "Integrated Security=True";

// or

SqlConnection myConnection = new SqlConnection(
    "Data Source=MUM0212CPU0558\\SQLEXPRESS;Initial Catalog=students;Integrated Security=True"
);
  
```

- The connection string is actually a series of distinct pieces of information (connection string property) separated by semicolons `;`
- **Connection String Properties**
 - Data source: Name of the server where the data source is located.
 - Initial catalog: Name of the database that this connection will be accessing (can be changed using `Connection.ChangeDatabase()`)
 - Integrated security: True if you want to connect to SQL Server by using the Windows user account that's running the web page code, else False for connecting by supplying a user ID and password for the database.
 - ConnectionTimeout: The number of seconds your code will wait before generating an error if it cannot establish a database connection (default = 15).
 - UserId: Name of the user configured in SQL server.
 - Password: Password for the given SQL Server User ID.

2. Making the Connection

- Before you can perform any database operations, you need to explicitly open your connection.
- `myConnection.Open();`
- After you use the `Open()` method, you have a live connection to your database.

3. Using the Select Command

- To actually retrieve data, you also need:
 - A SQL statement that selects the information you want
 - A `Command` object that executes said SQL statement
 - A `DataReader` or `DataSet` object to access the retrieved records
- Command objects represent SQL statements.
- To use a Command object:

1. You define it
2. Specify the SQL statement you want to use
3. Specify an available connection
4. Execute the command



Good practice: You should open your connection just before you execute your command and close it as soon as the command is finished.

```
SqlCommand myCommand = new SqlCommand();
myCommand.Connection = myConnection;
myCommand.CommandText = "SELECT * FROM emp";

// or

SqlCommand myCommand = new SqlCommand("SELECT * FROM emp", myConnection);
```

4. Using the DataReader

- `DataReader` allows you to quickly retrieve all your results.
- The `DataReader` uses a live connection and should be used quickly and then closed.
- It supports fast-forward-only read-only access to your results, which is needed when retrieving information.
- To create a `DataReader`, you use the `ExecuteReader()` method of the command object.
- After you have the reader, you retrieve a single row at a time by using the `Read()` method.
- You can then access the values in the current row by using the corresponding field names.
- To move to subsequent rows, use the `Read()` method again.
 - Return True if a row of information has been successfully retrieved.
 - Returns False if you attempted to read past the end of your result set.
- There is no way to move backward to a previous row.
- After reading all the results you need, close the `DataReader` and `Connection`

```
SqlDataReader myReader;
myReader = myCommand.ExecuteReader();
myReader.Read(); // Gets the first row in the result set
Console.WriteLine("Name: " + myReader["name"]); // Retrieve name column from the result set
myReader.Close();
myConnection.Close();
```

5. Updating Data


- You use the `Command` object, but you do not need a `DataReader` because no results will be retrieved.
- You use one of three SQL commands: Update, Insert, or Delete.
- To execute an Update, Insert, or Delete statement, you need to create a `Command` object.
- You can then execute the command with the `ExecuteNonQuery()` method.
- This method returns the number of rows that were affected, which allows you to check your assumptions.

```
//all options
SqlCommand myCommand = new SqlCommand("INSERT INTO emp VALUES('Gaurav', 69)", myConnection);
SqlCommand myCommand = new SqlCommand("UPDATE emp SET name='Gaurav' WHERE id=69", myConnection);
SqlCommand myCommand = new SqlCommand("DELETE FROM emp WHERE id=69", myConnection);
int rows = cmd.ExecuteNonQuery();
if (rows != 0) Console.WriteLine("Successful");
else Console.WriteLine("Not successful");
```

```
//all options
SqlCommand myCommand = new SqlCommand("INSERT INTO emp VALUES('Gaurav', 69)", myConnection);
SqlCommand myCommand = new SqlCommand("UPDATE emp SET name='Gaurav' WHERE id=69", myConnection);
SqlCommand myCommand = new SqlCommand("DELETE FROM emp WHERE id=69", myConnection);
int rows = cmd.ExecuteNonQuery();
```

```
if (rows != 0) Console.WriteLine("Successful");
else Console.WriteLine("Not successful");
```

Using Disconnected Data Access

 When you use disconnected data access, you use the DataSet to keep a copy of your data in memory.

- You connect to the database just long enough to fetch your data and dump it into the DataSet , then you disconnect immediately.
- **Reasons to use the DataSet to hold data in memory:**
 - The database connection is kept open for as little time as possible.
 - You want to use ASP.NET data binding to fill a web control with your data.
 - You want to navigate backward and forward through your data while you are processing it.
 - You want to navigate from one table to another.
 - You want to save the data to a file for later use. WriteXml() and ReadXml()
 - You want to store some data so it can be used for future requests (using caching).
- You fill the DataSet in pretty much the same way that you connect a DataReader .
- However, although the DataReader holds a live connection, information in the DataSet is always disconnected.

```
SqlCommand myCommand = new SqlCommand("SELECT * FROM emp", myConnection);
SqlDataAdapter adapter = new SqlDataAdapter(myCommand);
DataSet ds = new DataSet();
adapter.Fill(ds, "employees"); // Creates a new DataTable named employees inside the DataSet
foreach (DataRow row in ds.Tables["employees"].Rows)
{
    Console.WriteLine("Name: " + row["name"]); // Retrieve name column from the result set
}
myReader.Close();
myConnection.Close();
```

Connection Object (ADO)

Property	Description
CommandTimeout	Sets or returns a Long value that indicates, in seconds, how long to wait for a command to execute. Default is 30.
ConnectionTimeout	Sets or returns a Long value that indicates, in seconds, how long to wait for the connection to open. Default is 15.
ConnectionString	Connection string is used to establish and create connection to data source by using server name, database name, user id and password.
Mode	Sets or returns the access permissions in use by the provider on the current connection.
Provider	Sets or returns a String value that indicates the provider name.
State	Determine the current state of the connection at any time. (read-only)
Version	Returns the ADO version number. (read-only)

SqlCommand Class

Method	Description
ExecuteReader()	Returns data to the client as rows. This would typically be an SQL select statement or a Stored Procedure that contains one or more select statements. This method returns a DataReader object that can be used to fill a DataTable object or used directly for printing reports and so forth.
ExecuteNonQuery()	Executes a command that changes the data in the database, such as an update, delete, or insert statement, or a Stored Procedure that contains one or more of these statements. This method returns an integer that is the number of rows affected by the query.
ExecuteScalar()	This method only returns a single value. This kind of query returns a count of rows or a calculated value.

DataReader



Provides a high performance mechanism that reads a forward-only stream of rows from a data source.

- It retrieves the data from a data store as a data stream, while staying connected with the data source.

DataAdapter



The DataAdapter serves as a bridge between a `DataSet` and a data source for retrieving and saving data.

- The `Fill()` method adds or refreshes rows in the `DataSet` to match those in the data source.
- Useful when using data-bound controls in Windows Forms.
- Used to provide an easy way to manage the connection between your application and the underlying database tables, views and Stored Procedures.

DataSet

- The `DataSet` is the heart of ADO.NET.
- It is a collection of `DataTable` objects.
- Each `DataTable` object contains a collection of `DataColumn` and `DataRow` objects.
- The `DataSet` also contains a Relations collection that can be used to define relations among `DataTable` Objects.

Data Binding

- The basic principle of data binding is: You tell a control where to find your data and how you want it displayed, and the control handles the rest of the details.
- **Desktop or Client/Server Applications**
 - Data binding involves creating a direct connection between a data source and a control in an application window.
 - If the user changes a value in the onscreen control, the data in the linked database is modified automatically.
 - Similarly, if the database changes while the user is working with it, the display can be refreshed automatically.
 - This type of data binding isn't practical in the ASP.NET world, because you can't effectively maintain a database connection over the Internet.
 - This also severely limits scalability and reduces flexibility.
- **ASP.NET**
 - Data binding works in one direction only.
 - Information moves from a data object into a control.
 - Then the data objects are thrown away, and the page is sent to the client.
 - If the user modifies the data in a data-bound control, your program can update the corresponding record in the database (but nothing happens automatically).
 - It is much more flexible than old-style data binding.

Types of ASP.NET Data Binding

- Two types
 - Single-value
 - Repeated-value
- Data binding works a little differently depending on whether you're using single-value or repeated-value binding.
 - To use single-value binding, you must insert a data-binding expression into the markup in the `.aspx` file (not the code-behind file).
 - To use repeated-value binding, you must set one or more properties of a data control.
- After you specify data binding, you need to activate it by calling the `DataBind()` method.
- It automatically binds a control and any child controls that it contains.
- You can also bind the whole page at once by calling the `DataBind()` method of the current `Page` object.
- After you call this method, all the data-binding expressions are evaluated and replaced with the specified values in the page.
- Typically, this initialization is done when the `Page.Load()` event fires.

Single-Value or Simple Data Binding

- Used to add information anywhere on an ASP.NET page.
- You can even place information into a control property or as plain text inside an HTML tag.
- It allows you to take a variable, a property, or an expression and insert it dynamically into a page.
- To use it, you add special data-binding expressions into your `.aspx` files.
- `<%# expression %>` (Looks like a script block, but it isn't)
- The only thing you can add is a valid data-binding expression.
- **Drawbacks**
 - Putting code into a page's UI: One of ASP.NET's great advantages is that it allows developers to separate the UI code (`.aspx` file) from the actual code used for other tasks (in the codebehind file). The use of single-value data binding can encourage you to disregard that and start coding function calls and even operations into your UI page.
 - Fragmenting code: When using data-binding expressions, it may not be obvious where the functionality resides for different operations.
- **Examples**
 - A variable named Age: `<%# Age %>` (Must be declared as public, protected, or internal, but not private)
 - Built-in ASP.NET object `<%# Request.Browser.Browser %>`
 - Call functions: `<%# CalculateMarks(ID) %>`
 - Expressions: `<%# 1 + (69 * 420) %>`

```
protected void Page_Load(object sender, EventArgs e)
{
    // DB code for getting Marks
    Marks = 69; // Or hardcode the value
    URL = "Images/picture.jpg";
    // Now convert all the data-binding expressions on the page.
    this.DataBind();
}
```

```
<div>
    <asp:Label id="Label1" runat="server">
        Total marks: <%# Age %>
    </asp:Label>
    <asp:Image id="Image1" ImageUrl="<%# URL %>" runat="server" />
</div>
```



Data binding is a one-way method. Changing the variable after you've used the `DataBind()` method won't produce any visible effect. Unless you call the `DataBind()` method again, the displayed value won't be updated.

Repeated-Value or List Binding

- Repeated-value data binding allows you to display an entire table (or just a single field from a table).
- This requires a special control that supports it.
 - Web controls: `ListBox` `DropDownList` `CheckBoxList` `RadioButtonList`
 - Server-side HTML control: `HtmlSelect`
 - Rich web controls: `GridView` `DetailsView` `FormView` `ListView`
- A control supports repeated-value data binding if it provides a `DataSource` property.
- You can use also repeated-value binding to bind data from a collection or an array.
- When you call `DataBind()`, the control automatically creates a full list by using all the corresponding values.
- **Steps (Simple Data Binding)**
 1. Create and fill some kind of data object. like `array` `ArrayList` `Hashtable` `List` `Dictionary` `DataTable` `DataSet`
 2. Link the object to the appropriate control. You need to set a couple of properties, including `DataSource`. If you're binding to a full `DataSet`, you'll also need to set the `DataMember` property.
 3. You activate data binding by using the `DataBind()` method.
- You can bind the same data list object to multiple controls.
- **Dictionary Data Binding**
 - A dictionary collection is a special kind of collection in which every item is indexed with a specific key.

- Each item in a dictionary-style collection has both a key and a value associated with it.
- A `DataTextField` property adds the corresponding information to the `text` attribute in the control element.
- A `DataValueField` property adds the corresponding information to the `value` attribute in the control element.
- `MyListBox.DataTextField="Value"; MyListBox.DataValueField = "Key";`

- **ADO.NET Data Binding**

- `[Control].DataSource = cmd.ExecuteReader();`
- `[Control].DataTextField = "colname";`

Data Controls

- The rich data controls are quite a bit different from the simple list controls. For one thing, they are designed exclusively for data binding.
- They also have the ability to display more than one field at a time, often in a table-based layout or according to what you've defined.
- They also support higher-level features such as selecting, editing, and sorting.

The GridView

- The `GridView` is an extremely flexible grid control that displays a multicolumn table.
- Each record in your data source becomes a separate row in the grid.
- Each field in the record becomes a separate column in the grid.
- The `GridView` provides a `DataSource` property for the data object you want to display.
- Once you've set the `DataSource` property, you call the `DataBind()` method to perform the data binding and display each record in the `GridView`.
- However, it does not allow you to choose what column you want to display.
- It automatically generates a column for every field (if `AutoGenerateColumns` is True).

```
protected void Page_Load(object sender, EventArgs e)
{
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(sql, con);
    SqlDataAdapter adapter = new SqlDataAdapter(cmd);

    DataSet ds = new DataSet();
    adapter.Fill(ds, "Products");

    GridView1.DataSource = ds;
    GridView1.DataBind();
}
```

DetailsView

- The `DetailsView` displays a single record at a time.
- It places each field in a separate row of a table.
- Unfortunately, the `DetailsView` pager controls approach can be quite inefficient.
- One problem is that a separate postback is required each time the user moves from one record to another (whereas a grid control can show multiple records on the same page).
- Each time the page is posted back, the full set of records is retrieved, even though only a single record is shown.

FormView

- `FormView` provides a template-only control for displaying and editing a single record.
- The beauty of the `FormView` template model is that it matches quite closely the model of the `TemplateField` in the `GridView`.

XML TextWriter

- One of the simplest ways to create or read any XML document is to use the basic `XmlTextWriter` and `XmlTextReader` classes.
- First, you create or open the file.
- Then, you write to it or read from it, moving from top to bottom.
- Finally, you close it and get to work using the retrieved data in whatever way you'd like.
- Import the namespaces `using System.Xml;`
- To start a document, you always begin by calling `WriteStartDocument()`. To end it, you call `WriteEndDocument()`.

- You write the elements you need, in three steps.
 - First, you write the start tag by calling `WriteStartElement()`.
 - Then you write attributes, elements, and text content inside.
 - Finally, you write the end tag by calling `WriteEndElement()`.
- The methods you use always work with the current element.
- So if you call `WriteStartElement()` and follow it up with `WriteAttributeString()`, you are adding an attribute to that element.
- Similarly, if you use `WriteString()`, you insert text content inside the current element.
- If you use `WriteStartElement()` again, you write another element, nested inside the current element.

```
protected void Page_Load(object sender, EventArgs e)
{
    XmlTextWriter writer = new XmlTextWriter(
        "C:/Users/Admin/Documents/Visual Studio 2010/Projects/Practical7/Practical7/tvschedule.xml",
        System.Text.Encoding.UTF8
    );
    writer.Formatting = Formatting.Indented;
    writer.WriteStartDocument();
    // Root element
    writer.WriteStartElement("tvschedule");
    // Channel
    writer.WriteStartElement("channel");
    writer.WriteElementString("banner", "Banner123");
    // Day
    writer.WriteStartElement("day");
    writer.WriteElementString("date", "09th March 2024");
    // Programslot
    writer.WriteStartElement("programslot");
    writer.WriteElementString("time", "4:20PM");
    writer.WriteElementString("title", "El Clasico 2024");
    writer.WriteElementString("description", "Football go brrr");
    writer.WriteEndElement(); // programslot
    writer.WriteEndElement(); // day
    writer.WriteEndElement(); // channel
    writer.WriteEndElement(); // tvschedule
    writer.WriteEndDocument();
    writer.Close();
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<tvschedule>
  <channel>
    <banner>Banner123</banner>
    <day>
      <date>09th March 2024</date>
      <programslot>
        <time>4:20PM</time>
        <title>El Clasico 2024</title>
        <description>Football go brrr</description>
      </programslot>
    </day>
  </channel>
</tvschedule>
```

XML TextReader

- Reading the XML document in your code is just as easy with the corresponding `XmlTextReader` class.
- The `XmlTextReader` moves through your document from top to bottom, one node at a time.
- You call the `Read()` method to move to the next node.
- This method returns
 - true if there are more nodes to read
 - false once it has read the final node

- The current node is provided through the properties of the `XmlTextReader` class, such as `NodeType` and `Name`.
- A node is a designation that includes comments, whitespace, opening tags, closing tags, content, and even the XML declaration at the top of your file.

```
protected void Page_Load(object sender, EventArgs e)
{
    XmlTextReader reader = new XmlTextReader(
        "C:/Users/Admin/Documents/Visual Studio 2010/Projects/Practical7/Practical7/tvschedule.xml"
    );
    while (reader.Read())
    {
        if (reader.NodeType == XmlNodeType.Element)
            Label1.Text += "Element " + reader.Name + " started<br/>";
        else if (reader.NodeType == XmlNodeType.Text)
            Label1.Text += "" + reader.Value + "<br/>";
        else if (reader.NodeType == XmlNodeType.EndElement)
            Label1.Text += "Element " + reader.Name + " ended<br/>";
    }
    reader.Close();
}
```

Caching



Caching is a concept of retaining important data for just a short period of time.

- Caching is often used to store information that's retrieved from a database.
- A good caching strategy identifies the most frequently used pieces of data that are the most time-consuming to create and stores them.
- Cache data (or web pages) that are
 - expensive (in terms of time and resources)
 - used frequently

Types of Caching

- **Output Caching**
 - This is the simplest type of caching.
 - It stores a copy of the final rendered HTML page that is sent to the client.
 - The next client that submits a request for this page doesn't actually run the page.
 - Instead, the final HTML output is sent automatically.
 - The time that would have been required to run the page and its code is completely reclaimed.
 - Without query string: `<%@ OutputCache Duration="20" VaryByParam="None" %>` (20 seconds)
 - Twenty seconds may seem like a trivial amount of time, but in a high-volume site, it can make a dramatic difference.
 - By caching the page for 20 seconds, you limit database access for this page to three operations per minute.
 - Without caching, the page will try to connect to the database once for each client and could easily make dozens of requests in the course of 20 seconds.
- **Data Caching**
 - This is carried out manually in your code.
 - To use data caching, you store important pieces of information that are time consuming to reconstruct (such as a `DataSet` retrieved from a database) in the cache.
 - Other pages can check for the existence of this information and use it, thereby bypassing the steps ordinarily required to retrieve it.

Caching and the Query String

- **Process**
 1. You request a page without any query string parameter and receive page copy A.
 2. You request the page with the parameter `ProductID=1`. You receive page copy B.
 3. Another user requests the page with the parameter `ProductID=2`. That user receives copy C.
 4. Another user requests the page with `ProductID=1`. If the cached output B has not expired, it's sent to the user.

5. The user then requests the page with no query string parameters. If copy A has not expired, it's sent from the cache.

- Developers overemphasize on the dynamic information in real-time.
 - Caching enables us to use outdated data for a brief period of time.
 - If the page uses information from the current user's session to tailor the UI, full page caching isn't appropriate because the same page cannot be reused for requests from different users (fragment caching may help).
 - All query string variables: `<%@ OutputCache Duration="20" VaryByParam="*" %>`
 - Selected Query String Variables:
 - Requests with different `VaryByParam` parameters will be cached separately, but all other parameters will be ignored.
 - `<%@ OutputCache Duration="20" VaryByParam="ProductID" %>`
 - `<%@ OutputCache Duration="20" VaryByParam="ProductID;Type" %>`
-