Chapter 9
Using ADO.NET
Overview

- This chapter covers how to programmatically and declaratively work with data in databases.
- It covers:
  - accessing and modifying data within databases in ASP.NET using the classes of ADO.NET.
  - the codeless approach to accessing data using data source controls.
Introducing ADO.NET

- ADO.NET refers to the various classes in the .NET Framework that provide data access.
- ADO.NET is typically used to access relational databases.
  - can also be used to access other external data such as XML documents.
ADO.NET architecture
The ADO.NET data providers are a handful of .NET classes that are used for:
- connecting to a database,
- executing data commands,
- populating a DataSet,
- providing fast, forward-only, read-only access to data.

The four main classes within a data provider are:
- DbConnection
- DbCommand
- DbDataAdapter
- DbDataReader
ADO.NET data providers

- The key point about .NET data providers is:
  - Each database type has its own version of these classes.
- Because DbConnection, DbCommand, DbDataAdapter, and DbDataReader are abstract classes, each provider implements its own concrete version of these classes.
  - SqlConnection class defined in the SQL Server data provider
  - OracleConnection class defined in the Oracle data provider
ADO.NET namespaces

System.Data

System.Data.Common

System.Data.OleDb

System.Data.SqlClient

System.Data.Odbc

System.Data.OracleClient
## Abstract Base Classes of a Data Provider

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbCommand</td>
<td>Executes a data command, such as a SQL statement or a stored procedure.</td>
</tr>
<tr>
<td>DbConnection</td>
<td>Establishes a connection to a data source.</td>
</tr>
<tr>
<td>DbDataAdapter</td>
<td>Populates a <code>DataSet</code> from a data source.</td>
</tr>
<tr>
<td>DbDataReader</td>
<td>Represents a read only, forward-only stream of data from a data source.</td>
</tr>
</tbody>
</table>
Using Data Provider Classes

- The process for using these classes is as follows:
  1. Create a `DbConnection` to the database via a connection string.
  2. Create and execute a `DbCommand` for the database.
  3. [Optional] Fill a `DataSet` from the database using a `DbDataAdapter`, or use a `DbDataReader` to retrieve data from the database.
DBConnection

- Represents a connection to a data source through which commands are passed to the data source and through which data is returned.
- Before you can access a database, you must first create a connection to it.
- Database commands then “travel” across the connection to the database, as does any data returned from a database.
DBConnection

- Each DbConnection class has members for:
  - opening and closing a connection,
  - setting and retrieving properties of a connection,
  - handling connection-related events.
Connection Strings

- To create a connection, you must specify a **connection string**.
- A connection string is a string that specifies the initialization and connection information needed to connect to a data source.
- It can contain information such as the database name or location, driver type, user name, password, and even connection pooling and timeout information.
- Connection strings are based on ODBC connection string syntax consisting of name-value pairs separated by semicolons.

```
name1=value1; name2=value
```

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\data\abc.mdb;
```
To create a connection, you simply:

- instantiate the relevant `DbConnection` object (passing it your connection string)
- call its `Open` method.

The `DbConnection` object is now available for use.

After you no longer need the connection, you must close it.

```csharp
// Must use @ since string contains backslash characters
string connString = @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\data\abc.mdb;"

OleDbConnection conn = new OleDbConnection(connString);
conn.Open();

// Now use the connection
...

// all finished, so close connection
conn.Close();
```
Exception Handling

- In general, any time your pages need to access an external resource, it is possible that an exception could occur.
- As a consequence, you should wrap your connection opening code within a try...catch block.

```csharp
string connString = ...
OleDbConnection conn = new OleDbConnection(connString);
try
{
    conn.Open();
    // Now use the connection...
}
catch (OleDbException ex)
{
    // Handle or log exception
}
finally
{
    if (conn != null)
        conn.Close();
}
```
Alternate Syntax

➤ There is an alternate programming pattern for using a connection that uses the C# using statement.

➤ This statement defines a scope for a block of code.

➤ After the block of code is finished executing, any objects defined in the using statement are automatically disposed.

```csharp
using (OleDbConnection conn = new OleDbConnection(connString))
{
    conn.Open();

    // Now use the connection

    ...
}
```
Storing Connection Strings

- In ASP.NET 2.0, you can define connection strings for your application in the `Web.config` file.

```xml
<configuration>
  <connectionStrings>
    <add name="Books" connectionString="Provider=..." />
    <add name="Sales" connectionString="Data Source=..." />
  </connectionStrings>
</configuration>

string connString = WebConfigurationManager.ConnectionStrings["Books"].ConnectionString;
```
Connection Pooling

➢ Two key points:
  ➢ Database connections are a finite commodity in a DBMS.
  ➢ Creating and opening a connection is a relatively slow process.

➢ In a typical real-world Web site with many simultaneous requests, you can improve the performance and scalability of your application by reducing the number of times you connect to the database by sharing, or pooling, your connections to a data source.
Connection Pooling

- Connection pooling is handled automatically behind-the-scenes by the ADO.NET providers.
- For some of the providers you can customize how the pooling works or even turn it off, via various options in the connection string.
DbCommand

- Represents an SQL statement or a stored procedure that is executed by the data source.
- Each `DbCommand` class has members for:
  - representing an SQL statement,
  - creating data parameters,
  - executing SQL commands that either return data (e.g., SELECT) or do not return data (e.g., INSERT, DELETE, or UPDATE).
  - can also be used to run stored procedures if the database supports them.
To create a command, you instantiate the relevant `DbCommand` object and pass it your database `DbConnection` object.

This `DbConnection` can be set via the constructor or via the `DbConnection` property of the `DbCommand`.

The `DbCommand` object also needs a database command contained in a string.

```csharp
string connString = "...";
OleDbConnection conn = new OleDbConnection(connString);

// create a command using SQL text
string cmdString = "SELECT Id, Name, Price From Products";
OleDbCommand cmd = new OleDbCommand(cmdString, conn);

conn.Open();
...
Executing a DbCommand

Once a DbCommand object has been instantiated and its command text, command type, and connection set, the command can be run by calling one of the Execute methods of the DbCommand class:

- **ExecuteNonQuery**
  - for commands that do not return any record data, such as an SQL INSERT or DELETE.

- **ExecuteScalar**
  - for SELECT commands that return a single value.

- **ExecuteReader**
  - for SELECT commands that return multiple results.
  
- How are these results returned?
  - returns a DbDataReader.
Using DbParameters

- Represent a parameter to a command.
- Parameters are used to specify criteria to a query
  - that is, they are used to construct **parameterized queries**.
- They are a more secure alternative to simply building a query with criteria via string building.
Why String Building is Bad

- An example of a query constructed from string building is shown in the following example.
  - It uses the value entered by a user into a text box to construct the SQL query.

```csharp
// Do not do this
string sql = "SELECT * FROM Users WHERE User='" + txtUser.Text + "'";
```

- Why is this so bad?
  - This type of coding is vulnerable to the so-called **SQL injection attack**.
  - In this type of attack, a user enters SQL into a data entry form in order to extract additional information from the database that you want hidden or even to submit surreptitiously nasty commands to the DBMS.
What would be the resulting SQL string from the above code if the user entered the following into the text box?

' or 1=1 --

The resulting SQL string would be:

SELECT * FROM Users WHERE User='' OR 1=1 --'

Since two dashes indicate a SQL comment, the DBMS will ignore the final single quote.

This query will select all the records in the Users table, which might be problematic if the query results are being data bound to a control that can display multiple records.

What would happen if user entered:

'; DROP TABLE customers; --
Some of the dangers of SQL injection attacks can be avoided by using `DbParameter` objects to construct commands that use criteria constructed from user input.

A `DbParameter` is simply an object that represents a name-value pair,
- name is the identifier for the parameter
- value is the actual data for the parameter.
Using a DbParameter

Creating a command using a DbParameter involves three steps:

1. Modify the SQL WHERE clause so it uses parameter names instead of values.

   ```csharp
   string s = "select * from Users where UserId=@user";
   ```

2. Create the appropriate DbParameter objects and assign them the appropriate names and values.

   ```csharp
   SqlParameter param = new SqlParameter("@user",txtUser.Text);
   ```

3. Add the created DbParameter objects to the DbCommand object’s Parameters collection.

   ```csharp
   SqlCommand cmd;
   ...
   cmd.Parameters.Add(param);
   ```

   This step must be done before calling any of the DbCommand object’s Execute methods.
Transactions

- Transactions provide a way to gracefully handle errors and keep your data properly consistent when errors do occur.
- Transactions can be implemented both in ADO.NET as well as within the DBMS.
- Transactions can be **local** or **distributed**.
Local Transactions

- A local transaction affects operations that target a single database, and is ultimately mapped to the transaction mechanism within the DBMS.
Local Transaction Steps

- Create a DbTransaction object by calling the BeginTransaction method of the DbConnection class.

- Assign the DbTransaction object to each DbCommand object being executed as part of the transaction via its Transaction property.

- Execute each DbCommand.

- **Commit** (i.e., save the database changes) if everything worked okay or **roll back** (i.e., undo any database changes) if an exception occurred.

  - If the connection is closed before a commit or a roll back (caused, for instance, by a crash in the DBMS or the web server), then the transaction would be rolled back.
A distributed transaction is a transaction that affects not just a single database, but multiple resources.

For instance, in a typical order processing site, a distributed transaction might be involved since an order requires not only a local database, but also:

- the credit card processor,
- And external legacy ordering system.

Because there are multiple external resources involved, distributed transactions are much more complicated than local transactions.
Distributed transactions

- Distributed transactions are handled by classes in the `System.Transaction` namespace and may as well involve working with non-managed APIs such as Microsoft Message Queue (MSMQ) or COM+.
The DbDataReader is optimized for the fast retrieval of a read-only stream of records and is thus ideal for web applications.

DbDataReader is not a data container like the DataSet, but a kind of pointer to a record in a result set (that is, a set of records returned from a database query).

DbDataReader also implements the IEnumerable interface so multi-value web server controls can be data bound to it.
Programming a DbDataReader

- DbDataReader objects are not created using the C# `new` operator.
- Instead, you use the `ExecuteReader` method of `DbCommand`.

```csharp
string connString = "...";
OleDbConnection conn = new OleDbConnection(connString);

string cmdString = "SELECT Id,ProductName,Price From Products";
OleDbCommand cmd = new OleDbCommand(cmdString, conn);

conn.Open();
OleDbDataReader reader = cmd.ExecuteReader();

someControl.DataSource = reader;
someControl.DataBind();

reader.Close();
conn.Close();
```
If you are not using data binding, you can still use the `DbDataReader`.

If you will be processing multiple records, you will need to loop through the records.

You can use the `Read` method of `DbDataReader` to move the read cursor to the next record in the result set.

```csharp
OleDbDataReader reader = cmd.ExecuteReader();
while ( reader.Read() )
{
    // process the current record
}
reader.Close();
```
Field data can be retrieved from the current record in the reader since the reader object acts like a collection, with each element in the collection corresponding to a field in the record.

SELECT Id, ProductName, Price FROM Products

// retrieve using column name
int id = (int)reader["Id"];
string name = (string)reader["ProductName"];
double price = (double)reader["Price"];

// retrieve using a zero-based column ordinal
int id = (int)reader[0];
string name = (string)reader[1];
double price = (double)reader[2];

// retrieve a typed value using column ordinal
int id = reader.GetInt32(0);
string name = reader.GetString(1);
double price = reader.GetDouble(2);
DbDataAdapter

- These classes represent a bridge between the DataSet container and an underlying database.
- Each DbDataAdapter class provides a:
  - Fill method for filling a DataSet (or just a DataTable) with data from the database
  - Update method for outputting any changes made to the data in the Dataset back to the database.
- The DbDataAdapter can also persist changes made to the in-memory data by writing the changes back to the database.
Programming a DbDataAdapter

```csharp
DataSet ds = new DataSet();

// create a connection
SqlConnection conn = new SqlConnection(connString);

string sql = "SELECT Isbn,Title,Price FROM Books";
SqlCommand cmd = new SqlCommand(sql, conn);
SqlDataAdapter adapter = new SqlDataAdapter(sql, conn);

try{
    // read data into DataSet
    adapter.Fill(ds);

    // use the filled DataSet
}
catch (Exception ex)
{
    // process exception
}
```
Data provider-independent ADO.NET coding

- While the data provider architecture of ADO.NET has several advantages, it can have one substantial disadvantage:
  - the amount of programming work that would be required to switch to a different data provider.
  - i.e., replace each provider-dependent reference to a different provider.

```csharp
OleDbConnection conn = new OleDbConnection(connString);
...
OleDbCommand cmd = new OleDbCommand(cmdString, conn);
...
OleDbDataReader reader = cmd.ExecuteReader();
...

SqlConnection conn = new SqlConnection(connString);
...
SqlCommand cmd = new SqlCommand(cmdString, conn);
...
SqlDataReader reader = cmd.ExecuteReader();
...
Data provider-independent ADO.NET coding

- We can partially achieve this aim on the one hand by coding to the abstract base classes used by the specific provider classes.

```csharp
DbConnection conn = new SqlConnection(connString);
```
ADO.NET 2.0 has added the `DbProviderFactory` and `DbProviderFactories` classes as a way to instantiate provider objects in a provider-independent manner.

```csharp
string invariantName = "System.Data.OleDb";
DbProviderFactory factory = DbProviderFactories.GetFactory(invariantName);

DbConnection conn = factory.CreateConnection();
conn.ConnectionString = connString;

DbCommand cmd = factory.CreateCommand();
cmd.CommandText = "SELECT * FROM Authors";
cmd.Connection = conn;

// The invariant name could be read-in from the web.config file rather than hard-coded.
```
Data Source Controls

- With **data source controls** it is possible to manipulate external data with **no** programming code.
- Data source controls provide a declarative approach (i.e., use markup) for accessing external data.
  - They do not have a user interface.
  - Instead they declaratively encapsulate an external data source.
Data Source Controls

➢ There are five different data source controls that come with ASP.NET.
  ➢ AccessDataSource
    ➢ Provides simplified access to a Microsoft Access database.
  ➢ SqlDataSource
    ➢ Provides access to any database that uses SQL.
  ➢ ObjectDataSource
    ➢ Allows you to use your own custom class (such as a business object or a data access object) to access data.
  ➢ SiteMapDataSource
    ➢ Used by the site navigation controls to access the XML-based site map file.
  ➢ XmlDataSource
    ➢ Provides access to any XML document
you can think of these controls as a single view into a data source.

```xml
<asp:AccessDataSource ID="dsBooks" runat="server"
DataFile="App_Data/BookCatalogSystem.mdb"
SelectCommand="Select AuthorId,AuthorName from Authors" />

<asp:DropDownList ID="drpAuthors" runat="server"
DataSourceID="dsBooks" DataTextField="AuthorName" />

<asp:SqlDataSource ID="dsArt" runat="server"
ConnectionString="<%$ ConnectionStrings:Books %>
ProviderName="System.Data.SqlClient"
SelectCommand="Select AuthorId,AuthorName From Authors" />

<asp:DropDownList ID="drpAuthors" runat="server"
DataSourceID="dsArt" DataTextField="AuthorName" />
```
Using Parameters

- We may not always want to retrieve every record in a database table. In SQL we can use the WHERE clause to filter the retrieved records.

- We can accomplish the same thing with our data source controls by adding a nested SelectParameters element to our data source declaration.

  - This SelectParameters element can contain any number of parameter control definitions within it.

```xml
<asp:SqlDataSource ID="dsArt" runat="server" ConnectionString="<%$ ConnectionStrings:Books %>">
  SelectCommand="Select AuthorId,AuthorName From Authors" >
  <SelectParameters>
    Parameter control definitions go here
  </SelectParameters>
</asp:SqlDataSource>
```
Parameter Types

- In a web application the values for a parameter could come from a variety of sources.
- We might want to use a value from
  - another control on the form,
  - a value from a query string,
  - a value from a cookie or a session variable.
To support this need, there are a number of different parameter control types that we can use within the `SelectParameter` element.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlParameter</td>
<td>Sets the parameter value to a property value in another control on the page.</td>
</tr>
<tr>
<td>CookieParameter</td>
<td>Sets the parameter value to the value of a cookie.</td>
</tr>
<tr>
<td>FormParameter</td>
<td>Sets the parameter value to the value of a HTML form element.</td>
</tr>
<tr>
<td>ProfileParameter</td>
<td>Sets the parameter value to the value of a property of the current user profile.</td>
</tr>
<tr>
<td>QuerystringParameter</td>
<td>Sets the parameter value to the value of a query string field.</td>
</tr>
<tr>
<td>SessionParameter</td>
<td>Sets the parameter value to the value of an object currently stored in session state.</td>
</tr>
</tbody>
</table>
Using Parameters

```csharp
<asp:DropDownList ID="drpPublisher" runat="server"
    DataSourceID="dsPublisher" DataValueField="PublisherId"
    DataTextField="PublisherName" AutoPostBack="True"/>

...<asp:SqlDataSource ID="dsBooks" runat="server"
    ProviderName="System.Data.OleDb"
    ConnectionString="<%$ ConnectionStrings:Catalog %>">
    SelectCommand="SELECT ISBN, Title FROM Books WHERE PublisherId=@Pub"
    <SelectParameters>
    <asp:ControlParameter ControlID="drpPublisher"
        Name="Pub" PropertyName="SelectedValue" />
    </SelectParameters>
</asp:SqlDataSource>
```
How do they work?

- Data source controls allow you to perform common data retrieval tasks with no programming. But how do they work? What goes on behind the curtain?
  - The SQL and Access data source controls are merely a declarative encapsulation of the ADO.NET coding that was covered earlier.
  - While these controls eliminate the need for the typical repetitive ADO.NET programming code, there is some controversy about them among developers.
The main issue revolves around the suitability of having data access details in the presentation layer.

Many ASP.NET developers very much prefer to keep their web forms free of database implementation details such as connection strings, stored procedure names, or SQL statements.

The main reason for this preference is to improve maintainability.

A large web application with dozens or even hundreds of web pages each with multiple SqlDataSource controls will be very difficult to maintain if implementation details about data access ever needs to be changed.

Instead, developers often prefer to encapsulate database access into separate classes (often called data access objects) that will be used by the web forms or by other “higher-level” classes such as business objects.
Another potential issue with the SqlDataSource control is that its Select command is invoked every time the page is requested.

For a page with multiple postback possibilities, this is an unnecessary load on the DBMS.

One way to mitigate this problem is by setting the DataSourceMode to DataSet and then enable caching via the EnableCaching property.
Like the other data source controls, the **ObjectDataSource** control provides a codeless way to display and manipulate data.

Unlike with the **SqlDataSource** control, the **ObjectDataSource** control is not an intermediary for a database but for some other class.

Many developers prefer to design their web applications so that their web forms are only concerned with presentation details.

Other classes are used to implement data access logic, business logic, and application logic.

The **ObjectDataSource** allows developers to maintain this type of application architecture and keep the benefits of codeless data binding.
Imagine that we have a class named PublisherDA that contains a method named GetAll which returns a DataTable containing publisher data.

We could define an ObjectDataSource as follows.

```xml
<asp:ObjectDataSource ID="objCatalog" runat="server" SelectMethod="GetAll" TypeName="PublisherDA" />
```

The method defined by SelectMethod can return any other following:

- A DataTable, DataSet, or DataView.
- An object that implements IEnumerable.
- Any object.