ⓘ **Table of Contents**

Agenda

---

00. Type of modes 🔄

✓ **Connected Mode** 🔌 **To execute any query or execute procedure, open connection everytime then close the connected.**

Example: You want any transaction to immediately affect the database like banking apps

🖉 **Disconnected Mode** 📦 **Retrieve the data from the database and store it in an object. Then start working on that object independently from the database but remember to take those edits and update the database with the new object.**

Example: Results website - you want the result to be shown on the website and show it quickly to the user instead of every user querying his result from the database

---

01. What is ADO.NET 💻

📋 **Definition ADO.NET (ActiveX Data Objects .NET) is a set of classes in the .NET Framework that provides access to data sources such as SQL Server, Oracle, XML, and other databases. It's the primary data access technology for .NET applications.**

🌟 Key Features

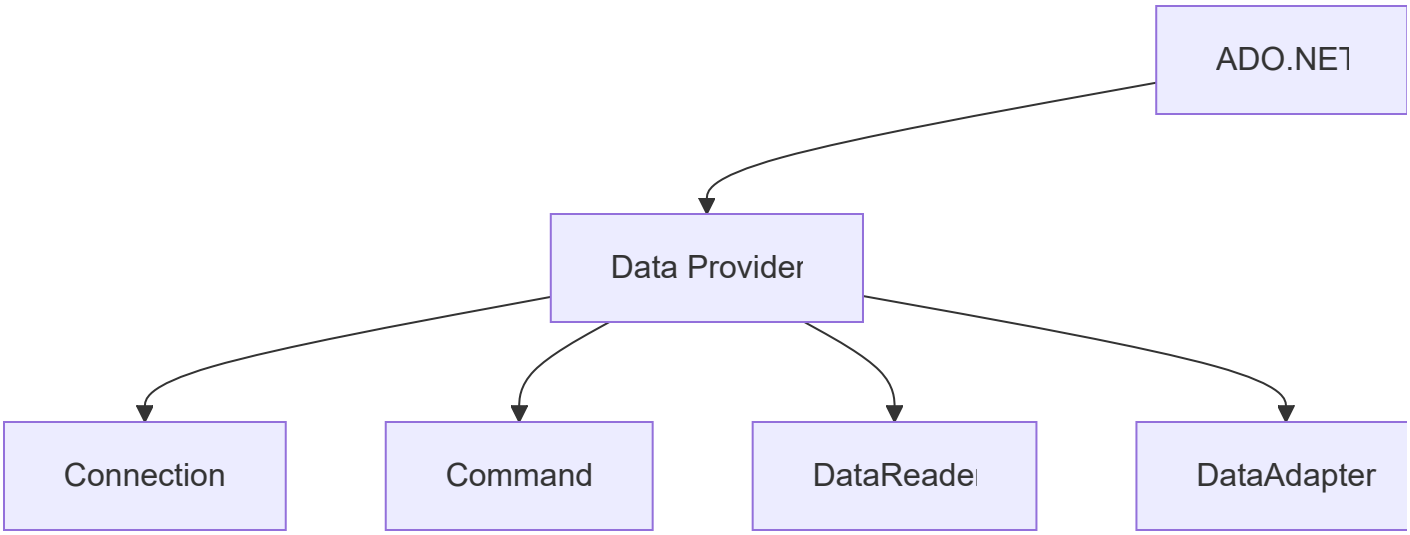
🔥 **Core Capabilities**

- ⚡ **Disconnected Architecture** - Works efficiently with disconnected data
- 📈 **Scalability** - Designed for enterprise-level applications
- 🔄 **Interoperability** - Works with XML for data exchange
- 🚀 **Performance** - Optimized for speed and efficiency
- 🔒 **Type Safety** - Strongly typed data access

---

02. Architecture Overview 🏗️

ⓘ **ADO.NET Structure ADO.NET architecture consists of two main components:**



1 Data Provider Components

Provider Components

- **Connection** - Establishes connection to data source
- **Command** - Executes commands against data source
- **DataReader** - Forward-only, read-only data stream
- **DataAdapter** - Bridge between DataSet and data source

2 DataSet Components

DataSet Components

- **DataSet** - In-memory cache of data
- **DataTable** - Represents a table of data
- **DataRelation** - Represents relationships between tables
- **DataView** - Customized view of a DataTable

---

03. Core Components

Connection Object

> Purpose The `Connection` object establishes a connection to a data source.

```
using System.Data.SqlClient; // For ADO .NET

// Connection string
///1. for user authentication
string connectionString = "Server=myServer;Database=myDB;User
Id=myUser;Password=myPass;";

///2. for windows authentication
string connectionString = "Server=myServer;Database=myDB;Integrated
Security=true;";

// Create connection
using (SqlConnection connection = new
SqlConnection(connectionString))
{
    connection.Open();
    Console.WriteLine("Connection opened successfully!");
    // Perform database operations
} // Connection automatically closed
```

ⓘ Connection String Components

- 🖥️ `Server` - Database server address
- 💿 `Database` - Database name
- 👤 `User Id` - Username for authentication
- 🔑 `Password` - Password for authentication
- 🪟 `Integrated Security=True` - For Windows Authentication

⚙️ Command Object

> Purpose The `Command` object executes SQL statements or stored procedures.

```
using (SqlConnection connection = new
SqlConnection(connectionString))
{
    connection.Open();

    SqlCommand command = new SqlCommand("SELECT * FROM Customers",
connection);

    // Execute command
    SqlDataReader reader = command.ExecuteReader();

    while (reader.Read())
    {
        Console.WriteLine($"{reader["CustomerID"]}:
{reader["CustomerName"]}");
    }

    reader.Close();
}
```

🔥 Command Methods

- 📖 `ExecuteReader()` - Returns DataReader for SELECT queries
- ✏️ `ExecuteNonQuery()` - For INSERT, UPDATE, DELETE (returns rows affected)
- 🎯 `ExecuteScalar()` - Returns single value (first column of first row)
- 📄 `ExecuteXML()` - Returns XML data

---

04. Connection Management 🔧

🏊 Connection Pooling

✓ **Performance Optimization ADO.NET automatically manages connection pooling for better performance.**

```
// Connection pooling is enabled by default
string connectionString =
    "Server=myServer;Database=myDB;" +
    "User Id=myUser;Password=myPass;" +
    "Min Pool Size=5;Max Pool Size=100;Connection Lifetime=60";
```

ⓘ **Pooling Parameters**

- ⬇️ `Min Pool Size` - Minimum number of connections in pool (default: 0)
- ⬆️ `Max Pool Size` - Maximum number of connections in pool (default: 100)
- ⏰ `Connection Lifetime` - Maximum age of connection (seconds)
- ❌ `Pooling=false` - Disable pooling

♻️ Using Statement Pattern

⚠ **Best Practice Always use `using` statements to ensure proper disposal:**

```
using (SqlConnection connection = new
SqlConnection(connectionString))
using (SqlCommand command = new SqlCommand(query, connection))
{
    connection.Open();
    // Work with database
} // Resources automatically disposed
```

---

05. Data Providers 🔌

🖉 **Available Providers ADO.NET includes several built-in data providers:**

Provider	Namespace	Description
🔷 SQL Server	`System.Data.SqlClient`	Optimized for SQL Server
🟠 OLE DB	`System.Data.OleDb`	Legacy databases
🟡 ODBC	`System.Data.Odbc`	ODBC data sources
🔴 Oracle	`System.Data.OracleClient`	Oracle databases

💡 Example: Different Providers

```
// SQL Server
using System.Data.SqlClient;
SqlConnection sqlConn = new SqlConnection(sqlConnectionString);

// Oracle
using System.Data.OracleClient;
OracleConnection oracleConn = new
OracleConnection(oracleConnectionString);

// OLE DB
using System.Data.OleDb;
OleDbConnection oleConn = new OleDbConnection(oleConnectionString);
```

---

06. Working with Commands 💬

🛡️ Parameterized Queries

⚡ **Security Critical** **Always use parameters** to prevent SQL injection:

```
string query = "SELECT * FROM Users WHERE Username = @username AND
Password = @password";

using (SqlConnection connection = new
SqlConnection(connectionString))
using (SqlCommand command = new SqlCommand(query, connection))
{
```

```
    // Add parameters
    command.Parameters.AddWithValue("@username", username);
    command.Parameters.AddWithValue("@password", password);

    connection.Open();
    SqlDataReader reader = command.ExecuteReader();

    if (reader.Read())
    {
        Console.WriteLine("Login successful!");
    }
}
```

📦 Stored Procedures

```
using (SqlConnection connection = new
SqlConnection(connectionString))
using (SqlCommand command =
        new SqlCommand("sp_GetCustomerOrders", connection))
{
    command.CommandType = CommandType.StoredProcedure;

    // Input parameter
    command.Parameters.AddWithValue("@CustomerID", customerId);

    // Output parameter
    SqlParameter outputParam =
         new SqlParameter("@TotalOrders", SqlDbType.Int);

    outputParam.Direction = ParameterDirection.Output;
    command.Parameters.Add(outputParam);

    connection.Open();
    command.ExecuteNonQuery();

    int totalOrders = (int)outputParam.Value;
    Console.WriteLine($"Total Orders: {totalOrders}");
}
```

📝 CRUD Operations

➕ INSERT

```
string insertQuery = "INSERT INTO Customers (Name, Email, Phone)
VALUES (@name, @email, @phone)";

using (SqlConnection connection = new
SqlConnection(connectionString))
using (SqlCommand command = new SqlCommand(insertQuery, connection))
{
    command.Parameters.AddWithValue("@name", "John Doe");
    command.Parameters.AddWithValue("@email", "john@example.com");
    command.Parameters.AddWithValue("@phone", "123-456-7890");

    connection.Open();
    int rowsAffected = command.ExecuteNonQuery();
    Console.WriteLine($"{rowsAffected} row(s) inserted.");
}
```

✏️ UPDATE

```
string updateQuery = "UPDATE Customers SET Email = @email WHERE
CustomerID = @id";

using (SqlConnection connection = new
SqlConnection(connectionString))
using (SqlCommand command = new SqlCommand(updateQuery, connection))
{
    command.Parameters.AddWithValue("@email",
"newemail@example.com");
    command.Parameters.AddWithValue("@id", 1);

    connection.Open();
    int rowsAffected = command.ExecuteNonQuery();
    Console.WriteLine($"{rowsAffected} row(s) updated.");
}
```

❌ DELETE

```
string deleteQuery = "DELETE FROM Customers WHERE CustomerID = @id";

using (SqlConnection connection = new
SqlConnection(connectionString))
using (SqlCommand command = new SqlCommand(deleteQuery, connection))
{
    command.Parameters.AddWithValue("@id", 1);

    connection.Open();
    int rowsAffected = command.ExecuteNonQuery();
    Console.WriteLine($"{rowsAffected} row(s) deleted.");
}
```

---

07. DataReader vs DataSet ⚖️

📖 DataReader (Connected Architecture)

✓ **Characteristics**

- ➡️ Forward-only, read-only
- ⚡ Faster performance
- 💾 Less memory overhead
- 🔌 Requires open connection
- 📊 Best for large datasets when you only need to read once

```
using (SqlConnection connection = new
SqlConnection(connectionString))
using (SqlCommand command = new SqlCommand("SELECT * FROM Products",
connection))
{
    connection.Open();
    SqlDataReader reader = command.ExecuteReader();

    while (reader.Read())
```

```
    {
        int id = reader.GetInt32(0);
        string name = reader.GetString(1);
        decimal price = reader.GetDecimal(2);

        Console.WriteLine($"{id}: {name} - ${price}");
    }

    reader.Close();
}
```

💾 DataSet (Disconnected Architecture)

🖊️ Characteristics

- ↔️ Can navigate forward and backward
- 📴 Works disconnected from database
- 💿 More memory intensive
- 🖇️ Supports multiple tables and relationships
- 🛠️ Best for complex data manipulation

```
using (SqlConnection connection = new
SqlConnection(connectionString))
{
    SqlDataAdapter adapter =
        new SqlDataAdapter("SELECT * FROM Products", connection);

    DataSet dataSet = new DataSet();

    adapter.Fill(dataSet, "Products");

    // Work with data offline
    DataTable table = dataSet.Tables["Products"];

    foreach (DataRow row in table.Rows)
    {
```

```
        Console.WriteLine($"{row["ProductID"]}: {row["ProductName"]}
- ${row["Price"]}");
    }

    // Modify data
    row["Price"] = 99.99;

    // Update database
    SqlCommandBuilder builder = new SqlCommandBuilder(adapter);
    adapter.Update(dataSet, "Products");
}
```

📊 Comparison Table

Feature	DataReader	DataSet
🔌 Connection	Connected	Disconnected
📖 Access	Forward-only	Random access
⚡ Speed	Faster	Slower
💾 Memory	Low	Higher
✏️ Updateable	No	Yes
📚 Multiple Tables	No	Yes

---

08. DataAdapter 🌉

> Purpose The `DataAdapter` serves as a bridge between DataSet and the database.

```
using (SqlConnection connection = new
SqlConnection(connectionString))
{
    // Create adapter with SELECT command
    SqlDataAdapter adapter = new SqlDataAdapter("SELECT * FROM
```

```
Customers", connection);

    // Create CommandBuilder to auto-generate INSERT, UPDATE, DELETE
commands
    SqlCommandBuilder builder = new SqlCommandBuilder(adapter);

    DataSet dataSet = new DataSet();
    adapter.Fill(dataSet, "Customers");

    // Modify data
    DataTable customersTable = dataSet.Tables["Customers"];
    DataRow newRow = customersTable.NewRow();
    newRow["Name"] = "Jane Smith";
    newRow["Email"] = "jane@example.com";
    customersTable.Rows.Add(newRow);

    // Update existing row
    customersTable.Rows[0]["Email"] = "updated@example.com";

    // Delete row
    customersTable.Rows[1].Delete();

    // Persist changes to database
    adapter.Update(dataSet, "Customers");
}
```

---

09. Working with Transactions 🔄

> 📋 **Definition Transactions ensure data integrity by grouping operations into atomic units.**

💳 Basic Transaction

```
using (SqlConnection connection = new
SqlConnection(connectionString))
{
    connection.Open();
    SqlTransaction transaction = connection.BeginTransaction();
```

```
    try
    {
        // First operation
        SqlCommand command1 =
         new SqlCommand(
         "UPDATE Accounts SET Balance = Balance -100 WHERE AccountID=
1",
          connection, transaction);

        command1.ExecuteNonQuery();

        // Second operation
        SqlCommand command2 =
         new SqlCommand(
         "UPDATE Accounts SET Balance = Balance +100 WHERE AccountID=
2",
         connection, transaction);

        command2.ExecuteNonQuery();

        // Commit if all succeed
        transaction.Commit();
        Console.WriteLine("Transaction completed successfully.");
    }
    catch (Exception ex)
    {
        // Rollback on error
        transaction.Rollback();
        Console.WriteLine($"Transaction failed: {ex.Message}");
    }
}
```

Transaction Isolation Levels

Isolation Levels

- `ReadUncommitted` - Lowest isolation, allows dirty reads

- 🟨 `ReadCommitted` - Default, prevents dirty reads
- 🟦 `RepeatableRead` - Prevents non-repeatable reads
- 🟩 `Serializable` - Highest isolation, prevents phantom reads
- 🟪 `Snapshot` - Uses row versioning

📋 Concurrency Issues Explained

🔴 Dirty Read

> ⚡ **Problem Reading uncommitted data from another transaction that might be rolled back.**

Example:

```
Transaction 1: UPDATE Accounts SET Balance = 1000 WHERE ID = 1
Transaction 2: SELECT Balance FROM Accounts WHERE ID = 1  -- Reads
1000
Transaction 1: ROLLBACK  -- Transaction 2 read data that never
existed!
```

🟡 Non-repeatable Read

> ⚠️ **Problem Reading the same row twice in a transaction but getting different values.**

Example:

```
Transaction 1: SELECT Balance FROM Accounts WHERE ID = 1  -- Reads
500
Transaction 2: UPDATE Accounts SET Balance = 1000 WHERE ID = 1
Transaction 2: COMMIT
Transaction 1: SELECT Balance FROM Accounts WHERE ID = 1  -- Reads
```

```
1000
-- Same query, different result!
```

🟢 Phantom Read

> ⚠ **Problem A query returns different sets of rows when executed twice in the same transaction.**

Example:

```
Transaction 1: SELECT * FROM Accounts WHERE Balance > 500  -- Returns
5 rows
Transaction 2: INSERT INTO Accounts VALUES (6, 600)
Transaction 2: COMMIT
Transaction 1: SELECT * FROM Accounts WHERE Balance > 500  -- Returns
6 rows
-- New "phantom" row appeared!
```

📊 Comparison Table

Level	Dirty Read	Non-repeatable Read	Phantom Read
🟥 ReadUncommitted	❌	❌	❌
🟨 ReadCommitted	✔	❌	❌
🟦 RepeatableRead	✔	✔	❌
🟩 Serializable	✔	✔	✔
🟪 Snapshot	✔	✔	✔

> ⓘ **Legend**
>
> - ✔ = Prevents this issue
> - ❌ = Does NOT prevent this issue

🔍 Detailed Explanation

1 ReadUncommitted

Characteristics

- **Lowest isolation level**
- Allows dirty reads, non-repeatable reads, and phantom reads
- Best performance but least data consistency
- Can read uncommitted changes from other transactions

2 ReadCommitted

Characteristics

- **Default isolation level in SQL Server**
- Prevents dirty reads
- Still allows non-repeatable reads and phantom reads
- Only reads committed data

3 RepeatableRead

Characteristics

- Prevents dirty reads and non-repeatable reads
- Still allows phantom reads
- Locks all data read by the transaction
- Same data will be read if queried again

4 Serializable

Characteristics

- **Highest isolation level**
- Prevents all concurrency issues
- Complete isolation from other transactions
- Lowest performance due to heavy locking
- Full data consistency

5 Snapshot

Characteristics

- Uses row versioning instead of locking
- Prevents dirty reads, non-repeatable reads, and phantom reads
- Better performance than Serializable
- Readers don't block writers, writers don't block readers

Setting Isolation Level

Different Isolation Levels

```
// Read Uncommitted - Fastest, least safe
SqlTransaction trans1 = connection.BeginTransaction(
    IsolationLevel.ReadUncommitted);

// Read Committed - Default, balanced
SqlTransaction trans2 = connection.BeginTransaction(
    IsolationLevel.ReadCommitted);

// Repeatable Read - More consistent
SqlTransaction trans3 = connection.BeginTransaction(
    IsolationLevel.RepeatableRead);

// Serializable - Most consistent, slowest
SqlTransaction trans4 = connection.BeginTransaction(
    IsolationLevel.Serializable);
```

```
// Snapshot - Row versioning approach
SqlTransaction trans5 = connection.BeginTransaction(
    IsolationLevel.Snapshot);
```

🎯 Choosing the Right Isolation Level

Scenario	Recommended Level	Reason
📊 High concurrency, reporting	ReadCommitted	Good balance
💰 Financial transactions	Serializable	Data accuracy critical
📖 Read-heavy workloads	Snapshot	No blocking
⚡ Quick dirty reads acceptable	ReadUncommitted	Maximum speed
🔒 Consistent reads required	RepeatableRead	Prevent changes

---

10. Best Practices ✅

1️⃣ Always Use Parameterized Queries

> ⚡ **Security Critical**

```
// ❌ BAD - SQL Injection vulnerable
string query = "SELECT * FROM Users WHERE Username = '" + username +
"'";

// ✅ GOOD - Safe from SQL injection
string query = "SELECT * FROM Users WHERE Username = @username";
command.Parameters.AddWithValue("@username", username);
```

2️⃣ Dispose Resources Properly

✓ **Best Practice**

```
// ✅ GOOD - Using statement ensures disposal
using (SqlConnection connection = new
SqlConnection(connectionString))
{
    // Work with connection
} // Automatically disposed
```

---

3 Use Connection Pooling

Performance

```
// ✅ Connection pooling enabled by default
// Don't disable unless necessary
```

---

4 Handle Exceptions Appropriately

⚠ **Error Handling**

```
try
{
    // Database operations
}
catch (SqlException ex)
{
    // Handle SQL-specific errors
    Console.WriteLine($"SQL Error: {ex.Message}");
    Console.WriteLine($"Error Number: {ex.Number}");
}
catch (Exception ex)
{
    // Handle general errors
```

```
    Console.WriteLine($"Error: {ex.Message}");
}
```

5️⃣ Choose Appropriate Data Access Method

🔥 Selection Guide

- 📖 Use **DataReader** for read-only, forward-only access
- 💾 Use **DataSet** when you need disconnected, complex data manipulation
- 📦 Use **stored procedures** for complex business logic

6️⃣ Optimize Performance

✓ Optimization

```
// Use CommandBehavior for optimization
using (SqlDataReader reader =
command.ExecuteReader(CommandBehavior.CloseConnection))
{
    // Read data
} // Connection closes automatically when reader is disposed
```

---

11. Common Patterns 🎨

📦 Repository Pattern

```
public interface ICustomerRepository
{
    Customer GetById(int id);
    IEnumerable<Customer> GetAll();
    void Add(Customer customer);
```

```
    void Update(Customer customer);
    void Delete(int id);
}

public class CustomerRepository : ICustomerRepository
{
    private readonly string _connectionString;

    public CustomerRepository(string connectionString)
    {
        _connectionString = connectionString;
    }

    public Customer GetById(int id)
    {
        using (SqlConnection connection = new
SqlConnection(_connectionString))
        using (SqlCommand command = new SqlCommand(
            "SELECT * FROM Customers WHERE CustomerID = @id",
connection))
        {
            command.Parameters.AddWithValue("@id", id);
            connection.Open();

            using (SqlDataReader reader = command.ExecuteReader())
            {
                if (reader.Read())
                {
                    return new Customer
                    {
                        CustomerID = reader.GetInt32(0),
                        Name = reader.GetString(1),
                        Email = reader.GetString(2)
                    };
                }
            }
        }
        return null;
    }

    public void Add(Customer customer)
```

```
    {
        using (SqlConnection connection = new
SqlConnection(_connectionString))
        using (SqlCommand command = new SqlCommand(
            "INSERT INTO Customers (Name, Email) VALUES (@name,
@email)", connection))
        {
            command.Parameters.AddWithValue("@name", customer.Name);
            command.Parameters.AddWithValue("@email",
customer.Email);

            connection.Open();
            command.ExecuteNonQuery();
        }
    }

    // Implement other methods...
}
```

🏗️ Data Access Layer

```
public class DataAccessLayer
{
    private readonly string _connectionString;

    public DataAccessLayer(string connectionString)
    {
        _connectionString = connectionString;
    }

    public DataTable ExecuteQuery(string query, Dictionary<string,
object> parameters = null)
    {
        using (SqlConnection connection = new
SqlConnection(_connectionString))
        using (SqlCommand command = new SqlCommand(query,
connection))
        {
```

```
            if (parameters != null)
            {
                foreach (var param in parameters)
                {
                    command.Parameters.AddWithValue(param.Key,
param.Value);
                }
            }

            SqlDataAdapter adapter = new SqlDataAdapter(command);
            DataTable table = new DataTable();
            adapter.Fill(table);
            return table;
        }
    }

    public int ExecuteNonQuery(string query, Dictionary<string,
object> parameters = null)
    {
        using (SqlConnection connection = new
SqlConnection(_connectionString))
        using (SqlCommand command = new SqlCommand(query,
connection))
        {
            if (parameters != null)
            {
                foreach (var param in parameters)
                {
                    command.Parameters.AddWithValue(param.Key,
param.Value);
                }
            }

            connection.Open();
            return command.ExecuteNonQuery();
        }
    }
}
```

12. Error Handling ⚠️

🔴 Common SQL Exceptions

```
try
{
    // Database operations
}
catch (SqlException ex)
{
    switch (ex.Number)
    {
        case 2:
            // Connection timeout
            Console.WriteLine("Connection timeout. Please check
server.");
            break;
        case 53:
            // Server not found
            Console.WriteLine("Server not found or not accessible.");
            break;
        case 2627:
            // Duplicate key (primary key violation)
            Console.WriteLine("Record already exists.");
            break;
        case 547:
            // Foreign key violation
            Console.WriteLine("Cannot delete record due to related
data.");
            break;
        default:
            Console.WriteLine($"SQL Error: {ex.Message}");
            break;
    }
}
```

---

13. Summary 📝

✓ **Key Takeaways ADO.NET provides a robust framework for data access in .NET applications.**

1. ✅ **Choose the right tool**: DataReader for speed, DataSet for flexibility
2. **Always parameterize**: Prevent SQL injection attacks
3. **Manage resources**: Use `using` statements for proper disposal
4. ⚠️ **Handle errors**: Implement proper exception handling
5. **Use transactions**: Ensure data integrity for related operations
6. **Follow patterns**: Implement repository or DAL patterns for maintainability

---

Author Information

Abdullah Ali

Contact: +201012613453

Remember ***"The key to mastering ADO.NET is understanding when to use connected vs disconnected architecture and always prioritizing security through parameterized queries."***