SqlCommand in ADO.NET (Connected Architecture)

Introduction

In ADO.NET, once a connection to the database has been successfully established using SqlConnection, the application still cannot interact with the database unless it can send instructions to it. Databases are passive systems; they only respond to explicit commands written in SQL.

The SqlCommand class represents these instructions in a .NET-compatible form. It acts as the communication medium through which a C# application sends SQL statements or stored procedure calls to SQL Server for execution.

In simple terms:

SqlCommand is the object that tells the database what to do.

Conceptual Role of SqlCommand

A SqlCommand object encapsulates three essential elements:

The SQL instruction or stored procedure name

The database connection on which the instruction will execute

Optional parameters that supply data safely to the command

The command does not manage the connection lifecycle. It assumes that the connection is already open in connected architecture. This separation of responsibility keeps the design clean and predictable.

From SQL Server's perspective, a SqlCommand corresponds to a request sent over an already established session. SQL Server parses the command, validates it, executes it, and then returns a result or acknowledgment.

Required Namespaces

To work with SqlCommand in a C# application, the following namespaces are required:

```
using System;
using System.Data;
using System.Data.SqlClient;
```

System.Data provides base ADO.NET abstractions

System.Data.SqlClient provides SQL Server-specific classes

Creating a SqlCommand Object

A SqlCommand object is usually created by passing the SQL instruction and the open SqlConnection.

```
string query = "SELECT * FROM Students";
```

```
SqlCommand command = new
SqlCommand(query, connection);
```

At this stage, no interaction with the database has occurred. The command is only configured. Execution happens only when one of the execution methods is called.

Executing Commands That Do Not Return Data (ExecuteNonQuery)

Some SQL statements modify data but do not return rows. Examples include INSERT, UPDATE, and DELETE.

For such operations, ExecuteNonQuery() is used.

```
string insertQuery =
"INSERT INTO Students (Name, Age) VALUES
('Amit', 22)";

using (SqlConnection connection = new
```

```
SqlConnection(connectionString))
{
connection.Open();

SqlCommand command = new
SqlCommand(insertQuery, connection);
int rowsAffected =
command.ExecuteNonQuery();

Console.WriteLine($"{rowsAffected} row(s)
inserted.");
}
```

ExecuteNonQuery() returns an integer representing the number of rows affected. This value is useful for validating whether the operation succeeded.

Executing Commands That Return a Single Value (ExecuteScalar)

When a query returns exactly one value (for example, aggregate results), ExecuteScalar()

is the most efficient method.

```
string countQuery = "SELECT COUNT(*) FROM
Students";

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

SqlCommand command = new
SqlCommand(countQuery, connection);
object result = command.ExecuteScalar();

int studentCount = Convert.ToInt32(result);
Console.WriteLine($"Total students:
{studentCount}");
}
```

ExecuteScalar() retrieves only the first column of the first row and ignores all other data. This makes it fast and memory-efficient.

Executing Commands That Return Multiple Rows (ExecuteReader)

When a query returns multiple rows, such as a SELECT statement, the data must be read sequentially using SqlDataReader.

```
string selectQuery = "SELECT Id, Name, Age
FROM Students";

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

SqlCommand command = new
SqlCommand(selectQuery, connection);
SqlDataReader reader =
command.ExecuteReader();

while (reader.Read())
{
int id = reader.GetInt32(0);
string name = reader.GetString(1);
```

```
int age = reader.GetInt32(2);

Console.WriteLine($"{id} {name} {age}");
}

reader.Close();
}
```

While the SqlDataReader is open:

The connection must remain open

Data is read forward-only

Data is read-only

This is a strict rule of connected architecture.

Using Parameters in SqlCommand (Security and Best Practice)

Embedding values directly into SQL queries is

unsafe and can lead to SQL Injection attacks. SqlCommand supports parameters to prevent this.

```
string query =
"SELECT * FROM Students WHERE Age > @age";

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

SqlCommand command = new
SqlCommand(query, connection);
command.Parameters.AddWithValue("@age", 18);

SqlDataReader reader =
command.ExecuteReader();

while (reader.Read())
{
Console.WriteLine(reader["Name"]);
}
```

```
reader.close();
}
```

Parameters ensure that user input is treated as data only, not executable SQL. This is a mandatory practice in real-world applications.

Executing Stored Procedures with SqlCommand

By default, SqlCommand assumes that the command text is a SQL query. To execute a stored procedure, the CommandType must be changed.

```
SqlCommand command = new
SqlCommand("GetAllStudents", connection);
command.CommandType =
CommandType.StoredProcedure;
```

Parameters can also be passed to stored procedures in the same way as SQL queries.

Relationship Between SqlCommand and SqlConnection

SqlCommand cannot execute without an open SqlConnection

Multiple commands can reuse the same connection

The connection controls where execution happens

The command controls what execution happens

This separation is fundamental to ADO.NET design.

Summary

SqlCommand is the core object that enables a C# application to interact with a SQL Server database. It represents SQL instructions, supports different execution modes, enables

secure parameterized queries, and works entirely on top of an open SqlConnection. Every data operation in connected architecture flows through SqlCommand.