

# 1 Lecture 5: Database Connectivity and CRUD in Windows Forms

**Course:** Scientific Visual Programming

**Topic:** Database Applications with C# Windows Forms

**Recommended duration:** 120 minutes

## 1.1 Learning objectives

Students should be able to explain CRUD operations, connect a Windows Forms application to a relational database, use parameterized commands, display records in a `DataGridView`, and design a basic insert–update–delete workflow.

### 1.2 1. Why use a database?

A file can be adequate for a small single-user application. A database becomes more appropriate when data must be searched, related, protected, shared, and updated reliably. A relational database stores data in tables and uses keys and relationships to maintain structure.

A Windows Forms application should not build SQL by concatenating user input. Parameterized commands protect against SQL injection and handle values more correctly.

### 1.3 2. CRUD operations

CRUD is an acronym for Create, Read, Update, and Delete. Create inserts a new record. Read retrieves records. Update changes an existing record. Delete removes a record, often after confirmation and sometimes through a soft-delete status.

A student table might contain `StudentId`, `FullName`, `Department`, and `Email`. `StudentId` is the primary key. The application should use the key to identify a specific row instead of assuming that a name is unique.

### 1.4 3. Connection and commands

The exact provider depends on the database. The following example uses SQL Server-style ADO.NET classes:

```
using Microsoft.Data.SqlClient;

string connectionString = "your-connection-string";
string sql = "SELECT StudentId, FullName, Department, Email FROM Students";

using var connection = new SqlConnection(connectionString);
using var adapter = new SqlDataAdapter(sql, connection);
var table = new DataTable();
adapter.Fill(table);
dataGridViewStudents.DataSource = table;
```

Store connection strings securely and avoid embedding passwords in source code. In a classroom project, configuration can be used; in production, secrets require an appropriate secure mechanism.

## 1.5 4. Parameterized insert

```
string sql = @"INSERT INTO Students (FullName, Department, Email)
            VALUES (@FullName, @Department, @Email)";

using var connection = new SqlConnection(connectionString);
using var command = new SqlCommand(sql, connection);
command.Parameters.AddWithValue("@FullName", txtName.Text.Trim());
command.Parameters.AddWithValue("@Department", cmbDepartment.Text);
command.Parameters.AddWithValue("@Email", txtEmail.Text.Trim());

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

In production code, choose parameter types and sizes explicitly when possible instead of relying on inferred types. Validate required fields before opening the connection.

## 1.6 5. Update and delete

An update command should use the selected primary key:

```
string sql = @"UPDATE Students
              SET FullName = @FullName, Email = @Email
              WHERE StudentId = @StudentId";
```

A delete operation should normally require confirmation and should verify that a row is selected. The application should refresh the displayed records after a successful change and clear the editor to prevent accidental reuse of stale values.

## 1.7 6. DataGridView workflow

The DataGridView can display a DataTable or a list of objects. When the user selects a row, copy the key and editable fields into the form controls. When the user clicks Save, decide whether the operation is insert or update based on whether a valid selected identifier exists.

Do not treat the displayed text alone as the source of truth. Keep the primary key in a hidden column or in a field associated with the selected record. Handle an empty result set with a clear message or an empty-state label.

## 1.8 7. Transactions and validation

A transaction groups several database operations so that they succeed or fail as one unit. For example, creating an order and its order lines should not leave an order without its lines if the second operation fails. Validation belongs at multiple levels. The form provides immediate feedback. The business layer applies domain rules. The database applies constraints such as required fields, unique values, and foreign keys. These layers complement one another.

## **1.9 8. Async and responsiveness**

Database operations can take time. A production application should avoid freezing the UI while waiting for a network database. Use asynchronous APIs and disable relevant commands while a request is in progress. Re-enable controls in a `finally` block so that an error does not leave the form permanently disabled.

## **1.10 9. Security and reliability**

Use least-privilege database accounts. Never display raw connection strings or sensitive exception details to end users. Log technical details securely and show a useful general message. Validate authorization on the server or data-access boundary; hiding a button is not sufficient security.

## **1.11 10. Practice activity**

Create a Student Management application connected to a relational database. Implement list, add, edit, delete, and search operations. Use parameterized SQL, validation, a `DataGridView`, confirmation before deletion, and a status label that reports success or failure. Document the table schema and identify the primary key.

## **1.12 Summary**

Database-driven Windows Forms applications combine controls, event handling, data access, validation, and security. CRUD operations should use primary keys and parameterized commands. A clean design separates presentation from business and data-access logic and keeps the user interface responsive.

## **1.13 References**