**1. Introduction**

The term **CRUD** stands for the four basic database operations - Create**, Read, Update, and Delete**.

In EF Core, CRUD operations are performed using the **DbContext** and **DbSet** classes, which represent your database and tables in object-oriented form.

|  |  |  |
| --- | --- | --- |
| **Operation** | **SQL Equivalent** | **EF Core Method** |
| Create | INSERT | Add(), AddAsync() |
| Read | SELECT | ToList(), Find(), Where() |
| Update | UPDATE | Update() |
| Delete | DELETE | Remove() |

**2. CRUD Architecture Overview**

In an ASP.NET Core Web API project, CRUD operations usually follow a **three-layer structure**:

1. **Model Layer** → Represents database entities.
2. **Data Access Layer (Repository)** → Contains logic for communicating with the database using EF Core.
3. **Controller Layer** → Exposes endpoints (API routes) that call repository methods.

This separation ensures the code is **modular**, **testable**, and **maintainable**.

**3. Read Operation (GET)**

The **Read** operation retrieves data from the database.

**Example:**

var students = \_context.Students.ToList();

* This fetches all rows from the Students table.
* EF Core translates it into:
* SELECT \* FROM Students;

**With Filter:**

var student = \_context.Students.FirstOrDefault(s => s.Id == id);

* This retrieves a specific student by ID.

**4. Create Operation (POST)**

The **Create** operation inserts new data into the database.

**Example:**

[HttpPost]

public async Task<IActionResult> AddStudent(Student student)

{

\_context.Students.Add(student);

await \_context.SaveChangesAsync();

return CreatedAtAction(nameof(GetStudentById), new { id = student.Id }, student);

}

**Explanation:**

* Add() → Adds the new record to the change tracker.
* SaveChangesAsync() → Executes the INSERT query.
* CreatedAtAction() → Returns HTTP 201 status code with the new resource URI.

**5. Update Operation (PUT)**

The **Update** operation modifies existing data.

**Example:**

[HttpPut("{id}")]

public async Task<IActionResult> UpdateStudent(int id, Student student)

{

if (id != student.Id)

return BadRequest("Student ID mismatch");

\_context.Entry(student).State = EntityState.Modified;

await \_context.SaveChangesAsync();

return Ok("Student updated successfully") }

**Explanation:**

* Marks entity as Modified in the DbContext.
* When SaveChangesAsync() is called, EF Core executes an UPDATE statement.

**6. Delete Operation (DELETE)**

The **Delete** operation removes a record.

**Example:**

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteStudent(int id)

{

var student = await \_context.Students.FindAsync(id);

if (student == null)

return NotFound("Student not found");

\_context.Students.Remove(student);

await \_context.SaveChangesAsync();

return Ok("Student deleted successfully");

}

**Explanation:**

* FindAsync() retrieves the record.
* Remove() marks it for deletion.
* SaveChangesAsync() performs the DELETE operation in SQL.

**7. Using Asynchronous Operations**

All EF Core operations can be executed asynchronously for better performance:

|  |  |
| --- | --- |
| **Operation** | **Async Method** |
| Add | AddAsync() |
| Save | SaveChangesAsync() |
| Find | FindAsync() |
| ToList | ToListAsync() |

Example:

var students = await \_context.Students.ToListAsync();

This prevents blocking the main thread during long-running database operations.

**8. Repository Pattern (Optional for Larger Projects)**

To keep the code modular, use a **repository pattern**.

**IStudentRepository.cs**

public interface IStudentRepository

{

Task<IEnumerable<Student>> GetAllAsync();

Task<Student> GetByIdAsync(int id);

Task<Student> AddAsync(Student student);

Task UpdateAsync(Student student);

Task DeleteAsync(int id);

}

**StudentRepository.cs**

public class StudentRepository : IStudentRepository

{

private readonly StudentDBContext \_context;

public StudentRepository(StudentDBContext context) => \_context = context;

public async Task<IEnumerable<Student>> GetAllAsync() => await \_context.Students.ToListAsync();

public async Task<Student> GetByIdAsync(int id) => await \_context.Students.FindAsync(id);

public async Task<Student> AddAsync(Student student)

{

\_context.Students.Add(student);

await \_context.SaveChangesAsync();

return student;

}

public async Task UpdateAsync(Student student)

{

\_context.Entry(student).State = EntityState.Modified;

await \_context.SaveChangesAsync();

}

public async Task DeleteAsync(int id)

{

var student = await \_context.Students.FindAsync(id);

if (student != null)

{

\_context.Students.Remove(student);

await \_context.SaveChangesAsync();

}

}

}

The controller can then depend on this interface instead of the direct context, improving **testability**.

**9. Error Handling and Validation**

Always validate and handle exceptions during CRUD operations:

**Example:**

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateException ex)

{

return BadRequest("Database update failed: " + ex.Message);

}

**Validation Example using Data Annotations:**

[Required]

[StringLength(50)]

public string Name { get; set; }

**10. HTTP Status Codes for CRUD APIs**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Method** | **Success Code** | **Description** |
| Create | POST | 201 | Resource created |
| Read | GET | 200 | Data retrieved |
| Update | PUT | 200 / 204 | Data updated |
| Delete | DELETE | 200 / 204 | Data deleted |

**11. Testing the API**

Once all methods are implemented:

* Launch Swagger (https://localhost:\*\*\*\*/swagger)
* Test:
  + **GET** → Fetch all students
  + **POST** → Add a new student
  + **PUT** → Update student
  + **DELETE** → Remove student

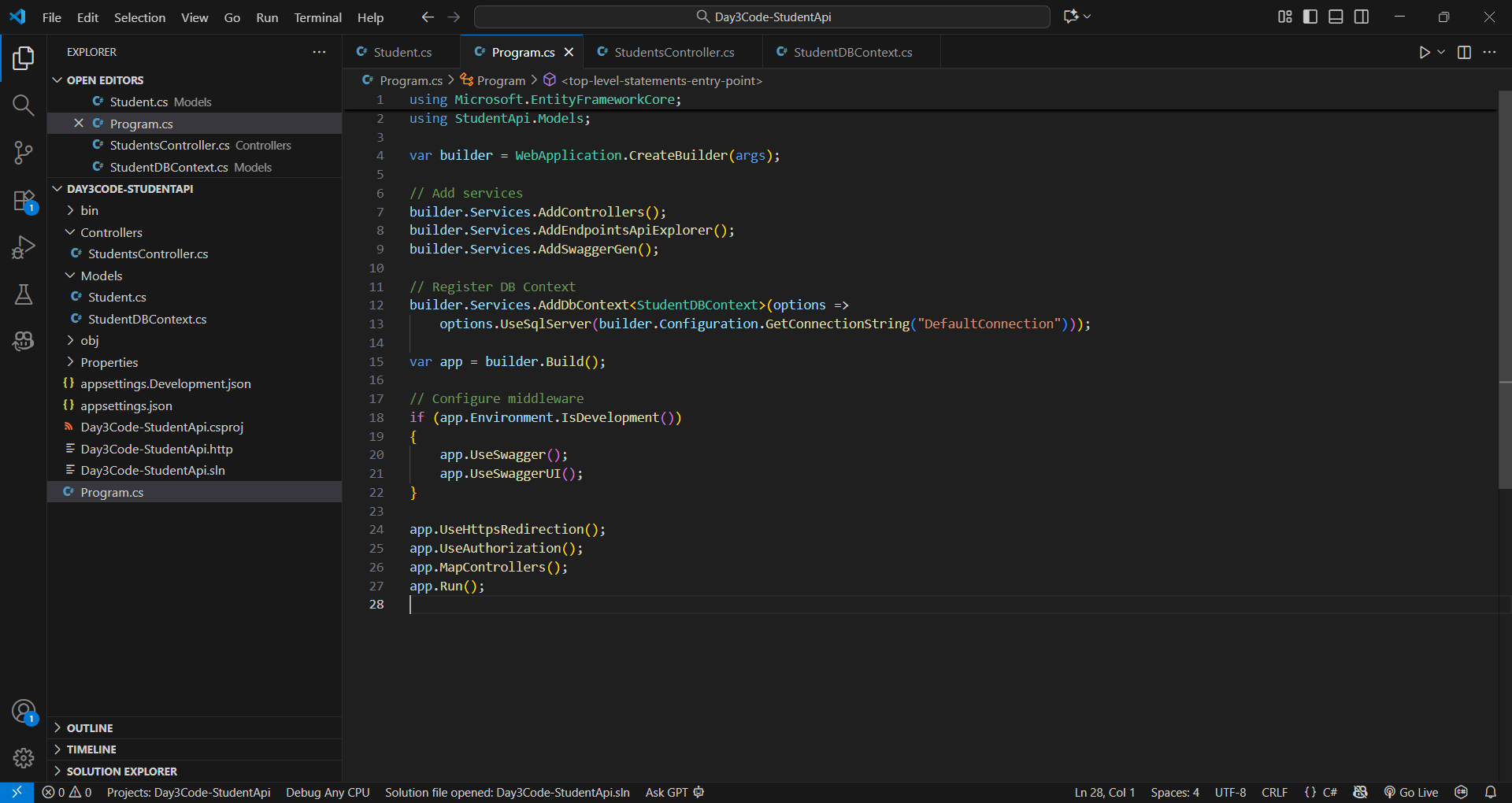
Take screenshots for documentation.

**Mini Task for Day 3**

**Build a Complete CRUD API for Students:**

* Use SQL Server database StudentDB.
* Create StudentController with:
  + GET /api/students
  + GET /api/students/{id}
  + POST /api/students
  + PUT /api/students/{id}
  + DELETE /api/students/{id}
* Test all operations in Swagger.

**Snapshots :**



Code : Program.cs

A screenshot of a computer

AI-generated content may be incorrect.

Code : Student.cs

A screenshot of a computer

AI-generated content may be incorrect.

Code : StudentDBContext.cs

A screenshot of a computer program

AI-generated content may be incorrect.

Code : StudentController.cs

A screenshot of a computer

AI-generated content may be incorrect.

Code : Database Code

A screenshot of a computer

AI-generated content may be incorrect.

Output : All method

A screenshot of a computer

AI-generated content may be incorrect.

Output : GET (All data from database)

A screenshot of a computer

AI-generated content may be incorrect.

Output : POST Method (Insert data )

A screenshot of a computer

AI-generated content may be incorrect.

Output : GET (Specific data from database)

A screenshot of a computer

AI-generated content may be incorrect.

Output : PUT (Update data )

A screenshot of a computer

AI-generated content may be incorrect.

Output : DELETE (Delete data )