**1. Objective**

Today’s goal is to improve your existing CRUD Web API by:

* Adding **input validation** using Data Annotations.
* Implementing **DTOs** to separate API models from database entities.
* Adding **global exception handling** and proper HTTP responses.

This ensures your backend is **secure**, **robust**, and **maintainable**.

**2. Why Validation, DTOs, and Error Handling Are Important**

|  |  |
| --- | --- |
| **Concept** | **Purpose** |
| **Validation** | Ensures that only correct and meaningful data enters the database. |
| **DTO (Data Transfer Object)** | Prevents over-posting and hides internal database structure from external users. |
| **Error Handling** | Prevents API crashes, provides user-friendly messages, and improves debugging. |

**Part 1 – Data Validation**

**3. What is Data Validation?**

Validation ensures that user input meets specific rules before being stored in the database.  
It prevents bad data, SQL injection, and runtime errors.

In ASP.NET Core, validation is usually done through **Data Annotations** in your model classes.

**4. Common Data Annotations**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Purpose** | **Example** |
| [Required] | Field must not be null or empty. | [Required] public string Name { get; set; } |
| [StringLength(50)] | Limits the string length. | [StringLength(50)] |
| [Range(1,100)] | Limits numeric range. | [Range(1,100)] |
| [EmailAddress] | Validates email format. | [EmailAddress] |
| [RegularExpression()] | Validates pattern. | [RegularExpression(@"^[A-Z]{2}\d{4}$")] |

**5. Example – Applying Validation to Model**

public class Student

{

[Key]

public int Id { get; set; }

[Required(ErrorMessage = "Name is required")]

[StringLength(50, ErrorMessage = "Name cannot exceed 50 characters")]

public string Name { get; set; }

[Range(1, 100, ErrorMessage = "Age must be between 1 and 100")]

public int Age { get; set; }

[StringLength(10)]

public string? Grade { get; set; }

}

If invalid data is sent, ASP.NET automatically returns **400 Bad Request** with a JSON error message.

**6. Handling Invalid Model Data**

You must always check ModelState.IsValid in the controller:

[HttpPost]

public async Task<IActionResult> AddStudent(Student student)

{

if (!ModelState.IsValid)

return BadRequest(ModelState);

\_context.Students.Add(student);

await \_context.SaveChangesAsync();

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

}

If validation fails, the API automatically returns which fields are invalid.

**Part 2 – Using DTOs (Data Transfer Objects)**

**7. What is a DTO?**

A **DTO** is a lightweight class used to transfer data between client and server.  
It hides unnecessary database details and protects sensitive fields.

For example, you might not want to expose fields like database IDs or internal metadata to the frontend.

**8. Why Use DTOs?**

|  |  |
| --- | --- |
| **Reason** | **Explanation** |
| **Security** | Prevents users from setting restricted fields like IDs. |
| **Clarity** | Ensures only necessary fields are visible in API responses. |
| **Maintainability** | Model changes won’t break client code. |

**9. Example – Creating a Student DTO**

**Create DTO Class (Models/DTOs/StudentDto.cs):**

namespace StudentApi.Models.DTOs

{

public class StudentDto

{

public string Name { get; set; }

public int Age { get; set; }

public string? Grade { get; set; }

}

}

Now, use it in your controller instead of exposing the full Student model.

**10. Using DTOs in Controllers**

[HttpPost]

public async Task<IActionResult> AddStudent(StudentDto studentDto)

{

if (!ModelState.IsValid)

return BadRequest(ModelState);

var student = new Student

{

Name = studentDto.Name,

Age = studentDto.Age,

Grade = studentDto.Grade

};

\_context.Students.Add(student);

await \_context.SaveChangesAsync();

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

}

**Part 3 – Error Handling in Web API**

**11. Why Error Handling is Required**

Without proper error handling:

* The app may crash on unexpected exceptions.
* Users might see unhandled technical messages.
* Debugging production issues becomes difficult.

Good error handling ensures **graceful failure** and **clear feedback**.

**12. Using Try-Catch in Controllers**

Example:

[HttpGet("{id}")]

public async Task<IActionResult> GetStudentById(int id)

{

try

{

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

if (student == null)

return NotFound("Student not found");

return Ok(student);

}

catch (Exception ex)

{

return StatusCode(500, $"Internal server error: {ex.Message}");

}

}

**Explanation:**

* try → executes normal logic.
* catch → catches exceptions (e.g., database errors).
* Returns status code **500 Internal Server Error** with a custom message.

**13. Using Global Error Handling (Middleware)**

Instead of repeating try-catch in every controller, you can create a **middleware** that catches all exceptions globally.

**Middleware Example:**

public class GlobalExceptionMiddleware

{

private readonly RequestDelegate \_next;

public GlobalExceptionMiddleware(RequestDelegate next)

{

\_next = next;

}

public async Task InvokeAsync(HttpContext context)

{

try

{

await \_next(context);

}

catch (Exception ex)

{

context.Response.StatusCode = 500;

await context.Response.WriteAsJsonAsync(new

{

Message = "An unexpected error occurred.",

Details = ex.Message

});

}

}

}

**Register in Program.cs:**

app.UseMiddleware<GlobalExceptionMiddleware>();

This handles all unhandled exceptions in one place and keeps controllers clean.

**14. Standard API Response Structure**

For production APIs, always return consistent responses:

{

"success": true,

"message": "Student updated successfully",

"data": {

"id": 1,

"name": "Udaya Kumar Shetty",

"age": 22,

"grade": "A"

}

}

This helps frontend developers parse responses uniformly.

**Mini Task for Day 4**

**Objective:**  
Upgrade your existing StudentApi to include:

1. Validation in the Student model.
2. Use of StudentDto for POST and PUT endpoints.
3. Add try-catch for all controller methods.
4. Optional: Implement global exception middleware.

**Snapshots :**

A screen shot of a computer

AI-generated content may be incorrect.

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

AI-generated content may be incorrect.

Code : StudentDto.cs

A screen shot of a computer program

AI-generated content may be incorrect.

Code : StudentController.cs

A screen shot of a computer

AI-generated content may be incorrect.

Code : GlobalExceptionMiddleware.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 )