Implementing Error Handling and Logging

Implementing Error Handling and Logging in ASP.NET Core

Objective: by the end of this activity, you will be able to implement error handling mechanisms in an ASP.NET Core web API project, set up logging using both built-in and third-party logging providers, and capture and analyze log data.

Step 1: Prepare for the Application

Create a small ASP.NET Core web API project in Visual Studio Code. This application will have a simple endpoint and include configurations for error handling and logging.

Steps:

1. Open Visual Studio Code and create a new ASP.NET Core web API project.
2. Remove any default code in the Program.cs file and test whether your environment is ready to start coding by verifying the project builds without errors.

Step 2: Basic Error Handling with Try-Catch Blocks

In this step, you'll add error handling to manage unexpected situations within an API endpoint.

Steps:

1. In the root of your project, create a new folder named Controllers.
2. Inside the Controllers folder, create a new file named ErrorHandlingController.cs.
3. In ErrorHandlingController.cs, define a new class named ErrorHandlingController and inherit from ControllerBase.
4. Add an action method named GetDivisionResult that takes two parameters, int numerator and int denominator.
5. Implement a try-catch block within GetDivisionResult to handle any division errors.
6. In the catch block, log a message to the console stating, "Error: Division by zero is not allowed," and return a user-friendly error message to the client.

Step 3: Global Error Handling Using Middleware

ASP.NET Core allows for centralized error handling with custom middleware. This step will guide you through setting up middleware to handle unhandled exceptions globally.

Steps:

1. Open the Program.cs file and add a new middleware using app.Use to handle exceptions globally.
2. Within this middleware, add a try-catch block that catches any unhandled exceptions.
3. Log a generic error message, and set the StatusCode to 500 for any errors captured in this middleware.
4. In the response, display a user-friendly message, such as "An unexpected error occurred. Please try again later."

Step 4: Setting Up Logging

Now, configure logging to capture application events and error details.

Steps:

1. In Program.cs, configure logging by setting up log levels (e.g., Error, Warning, Information).
2. Set up logging to output to the console by adding ConsoleLogger to the ILoggingBuilder.

Define log levels in appsettings.json for both Development and Production environments to control verbosity.

Step 5: Adding Third-Party Logging

Third-party logging frameworks like Serilog offer enhanced logging features. Here, you'll integrate Serilog into your project.

Steps:

1. Add Serilog to your project by installing the Serilog NuGet package.
2. Configure Serilog to write logs to a file and the console by setting up a new configuration in appsettings.json.

3. Update Program.cs to use Serilog as the logging provider by calling Log.Logger before building the host.

Step 6: Capturing and Analyzing Log Data

After setting up logging, test and analyze the generated logs to understand the application's behavior.

Steps:

1. Run the application and intentionally trigger errors (e.g., by calling GetDivisionResult with a denominator of zero).
2. Open the console and verify that the error logs contain appropriate messages and details.
3. Review the logged messages to ensure they match the format and detail level defined in previous steps.
4. Adjust log levels in appsettings.json if needed, and rerun the application to validate your changes.

ErrorHandlingController.cs:

```
using Microsoft.AspNetCore.Mvc;

namespace ErrorHandlingDemo.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ErrorHandlingController : ControllerBase
    {
        private readonly ILogger<ErrorHandlingController> _logger;

        public ErrorHandlingController(ILogger<ErrorHandlingController>
logger)
        {
            _logger = logger;
        }

        [HttpGet("divide")]
        public IActionResult GetDivisionResult(int numerator, int
denominator)
        {
            try
            {
                int result = numerator / denominator;
                _logger.LogInformation("Division succeeded: {Numerator}
/ {Denominator} = {Result}",
                    numerator, denominator, result);
```

```
                return Ok(new { Result = result });
            }
            catch (DivideByZeroException)
            {
                _logger.LogError("Division by zero is not allowed.
Numerator: {Numerator}", numerator);
                return BadRequest(new { Error = "Division by zero is not
allowed." });
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Unexpected error occurred during
division.");
                return StatusCode(500, new { Error = "An unexpected
server error occurred." });
            }
        }
    }
}
```

Program.cs:

```
using Serilog;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Host.UseSerilog((ctx, lc) => lc
    .ReadFrom.Configuration(ctx.Configuration)
    .WriteTo.Console());

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "ErrorHandlingDemo API",
        Version = "v1"
    });
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "ErrorHandlingDemo
API v1");
    });

    app.MapGet("/", context =>
    {
        context.Response.Redirect("/swagger");
        return Task.CompletedTask;
```

```
    });
}

app.Use(async (context, next) =>
{
    var logger =
context.RequestServices.GetRequiredService<ILogger<Program>>();
    try
    {
        await next();
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "Unhandled exception in global
middleware.");
        context.Response.StatusCode = 500;
        context.Response.ContentType = "application/json";
        await context.Response.WriteAsJsonAsync(new
        {
            Error = "An unexpected error occurred. Please try again
later."
        });
    }
});

app.MapControllers();
app.Run();
```

Appsettings.json:

```
{
  "AllowedHosts": "*",
  "Serilog": {
    "Using": [ "Serilog.Sinks.Console", "Serilog.Sinks.File" ],
    "MinimumLevel": {
      "Default": "Information",
      "Override": {
        "Microsoft": "Information",
        "System": "Warning"
      }
    },
    "WriteTo": [
      { "Name": "Console" },
      {
        "Name": "File",
        "Args": {
          "path": "logs/log-.txt",
          "rollingInterval": "Day",
          "outputTemplate": "[{Timestamp:yyyy-MM-dd HH:mm:ss}
{Level:u3}] {Message:lj}{NewLine}{Exception}"
        }
      }
    ]
  }
}
```