**Creating Microservices Architecture**

We are creating a very simple Microservices Architecture project

1. Create the projects as below:
   1. 2 .Net core Web API Project (Project names: UserService, Inventory)
   2. 1 ASP.Net core Web App (Razor Pages) project (Project name: UI)
2. Install these components in UserService Project

* Microsoft.EntityFrameworkCore
* Microsoft.EntityFrameworkCore.SqlServer
* Microsoft.EntityFrameworkCore.Tools

1. Codes for UserService Project

**Model**

**namespace UserService.Models**

namespace UserService.Models

{

public class User

{

public int Id { get; set; }

public string? Name { get; set; }

public string? Email { get; set; }

}

}

**DBContext**

using Microsoft.EntityFrameworkCore;

using UserService.Models;

namespace UserService.Data

{

public class UserDbContext : DbContext

{

public UserDbContext(DbContextOptions<UserDbContext> options) : base(options) { }

public DbSet<User> Users { get; set; };

}

}

**appSettings.json**

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"UserDb": "Server=localhost;Database=UserServiceDb;Trusted\_Connection=True;TrustServerCertificate=True;"

}

}

**Program.cs**

using Microsoft.EntityFrameworkCore;

using UserService.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

// Add DB Context

builder.Services.AddDbContext<UserDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("UserDb")));

// Add services

builder.Services.AddControllers();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

**Controller**

using Microsoft.AspNetCore.Mvc;

using UserService.Data;

using UserService.Models;

namespace UserService.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class UsersController : ControllerBase

{

private readonly UserDbContext \_context;

public UsersController(UserDbContext context)

{

\_context = context;

}

[HttpGet]

public IActionResult GetAll()

{

var users = \_context.Users.ToList();

return Ok(users);

}

[HttpGet("{id}")]

public IActionResult GetById(int id)

{

var user = \_context.Users.Find(id);

if (user == null) return NotFound();

return Ok(user);

}

[HttpPost]

public IActionResult Create(User user)

{

\_context.Users.Add(user);

\_context.SaveChanges();

return CreatedAtAction(nameof(GetById), new { id = user.Id }, user);

}

[HttpPut("{id}")]

public IActionResult Update(int id, User updatedUser)

{

var user = \_context.Users.Find(id);

if (user == null) return NotFound();

user.Name = updatedUser.Name;

user.Email = updatedUser.Email;

\_context.SaveChanges();

return NoContent();

}

[HttpDelete("{id}")]

public IActionResult Delete(int id)

{

var user = \_context.Users.Find(id);

if (user == null) return NotFound();

\_context.Users.Remove(user);

\_context.SaveChanges();

return NoContent();

}

}

}

Run migration script from Nuget Package Manager 🡪 Package Manager Console

Add-Migration Initial

Update-Database

1. Codes **for InventoryServices**:

NOTE: In the Inventory Services we are following a good structure.

**Models**

namespace InventoryService.Models

{

public class InventoryItem

{

public int Id { get; set; }

public string ProductName { get; set; } = string.Empty;

public int Quantity { get; set; }

public decimal Price { get; set; }

}

}

**DBContext**

using Microsoft.EntityFrameworkCore;

using InventoryService.Models;

namespace InventoryService.Data

{

public class InventoryDbContext : DbContext // Fix CS0311 by inheriting from DbContext

{

public InventoryDbContext(DbContextOptions<InventoryDbContext> options): base(options)

{

}

public DbSet<InventoryItem> Inventories { get; set; }

}

}

**appSettings.json**

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=InventoryDb;Trusted\_Connection=True;TrustServerCertificate=True;"

}

}

**Interface folders**

// Interfaces/IInventoryRepository.cs

using InventoryService.Models;

namespace InventoryService.Interfaces

{

public interface IInventoryRepository

{

Task<IEnumerable<InventoryItem>> GetAllAsync();

Task<InventoryItem?> GetByIdAsync(int id);

Task<InventoryItem> AddAsync(InventoryItem item);

Task<InventoryItem?> UpdateAsync(InventoryItem item);

Task<bool> DeleteAsync(int id);

}

}

// Interfaces/IInventoryService.cs

using InventoryService.Models;

namespace InventoryService.Interfaces

{

public interface IInventoryService

{

Task<IEnumerable<InventoryItem>> GetAllAsync();

Task<InventoryItem?> GetByIdAsync(int id);

Task<InventoryItem> AddAsync(InventoryItem item);

Task<InventoryItem?> UpdateAsync(InventoryItem item);

Task<bool> DeleteAsync(int id);

}

}

**Repositories Folder**

// Repositories/InventoryRepository.cs

using InventoryService.Data;

using InventoryService.Interfaces;

using InventoryService.Models;

using Microsoft.EntityFrameworkCore;

namespace InventoryService.Repositories

{

public class InventoryRepository : IInventoryRepository

{

private readonly InventoryDbContext \_context;

public InventoryRepository(InventoryDbContext context)

{

\_context = context;

}

public async Task<IEnumerable<InventoryItem>> GetAllAsync()

{

return await \_context.Inventories.ToListAsync();

}

public async Task<InventoryItem?> GetByIdAsync(int id)

{

return await \_context.Inventories.FindAsync(id);

}

public async Task<InventoryItem> AddAsync(InventoryItem item)

{

\_context.Inventories.Add(item);

await \_context.SaveChangesAsync();

return item;

}

public async Task<InventoryItem?> UpdateAsync(InventoryItem item)

{

var existing = await \_context.Inventories.FindAsync(item.Id);

if (existing == null) return null;

existing.ProductName = item.ProductName;

existing.Quantity = item.Quantity;

await \_context.SaveChangesAsync();

return existing;

}

public async Task<bool> DeleteAsync(int id)

{

var item = await \_context.Inventories.FindAsync(id);

if (item == null) return false;

\_context.Inventories.Remove(item);

await \_context.SaveChangesAsync();

return true;

}

}

}

**Services folder**

// Services/InventoryService.cs

using InventoryService.Interfaces;

using InventoryService.Models;

namespace InventoryService.Services

{

public class InventoryService : IInventoryService

{

private readonly IInventoryRepository \_repo;

public InventoryService(IInventoryRepository repo)

{

\_repo = repo;

}

public Task<IEnumerable<InventoryItem>> GetAllAsync() => \_repo.GetAllAsync();

public Task<InventoryItem?> GetByIdAsync(int id) => \_repo.GetByIdAsync(id);

public Task<InventoryItem> AddAsync(InventoryItem item) => \_repo.AddAsync(item);

public Task<InventoryItem?> UpdateAsync(InventoryItem item) => \_repo.UpdateAsync(item);

public Task<bool> DeleteAsync(int id) => \_repo.DeleteAsync(id);

}

}

**Controllers folder**

// Controllers/InventoryController.cs

using InventoryService.Interfaces;

using InventoryService.Models;

using Microsoft.AspNetCore.Mvc;

namespace InventoryService.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class InventoryController : ControllerBase

{

private readonly IInventoryService \_service;

public InventoryController(IInventoryService service)

{

\_service = service;

}

[HttpGet]

public async Task<IActionResult> GetAll()

{

var items = await \_service.GetAllAsync();

return Ok(items);

}

[HttpGet("{id}")]

public async Task<IActionResult> GetById(int id)

{

var item = await \_service.GetByIdAsync(id);

return item is null ? NotFound() : Ok(item);

}

[HttpPost]

public async Task<IActionResult> Add([FromBody]InventoryItem item)

{

var added = await \_service.AddAsync(item);

return CreatedAtAction(nameof(GetById), new { id = added.Id }, added);

}

[HttpPut("{id}")]

public async Task<IActionResult> Update(int id,[FromBody]InventoryItem item)

{

if (id != item.Id) return BadRequest("Mismatched ID");

var updated = await \_service.UpdateAsync(item);

return updated is null ? NotFound() : Ok(updated);

}

[HttpDelete("{id}")]

public async Task<IActionResult> Delete(int id)

{

var deleted = await \_service.DeleteAsync(id);

return deleted ? NoContent() : NotFound();

}

}

}

**Program.cs**

using InventoryService.Data;

using InventoryService.Interfaces;

using InventoryService.Repositories;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

builder.Services.AddDbContext<InventoryDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddScoped<IInventoryRepository, InventoryRepository>();

builder.Services.AddScoped<IInventoryService, InventoryService.Services.InventoryService>();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

So we are following the current structure as:

**Client → Controller → Service Interface → Service Implementation**

**→ Repository Interface → Repository Implementation → DbContext**

For our above implementation

Client 🡪 Controller → IInventoryService → InventoryService → IInventoryRepository → InventoryRepository → DbContext

1. Validating InventoryService using **FluentValidation** (just testing one method for the sake of learning)

**Install FluentValidation from Nuget Package**

**FluentValidation** (This package by jskinner. There are other packages also, don’t install them)

**Create a Validator**

Create a new folder Validators and add a class called InventoryItemValidator.cs

using FluentValidation;

using InventoryService.Models;

public class InventoryItemValidator : AbstractValidator<InventoryItem>

{

public InventoryItemValidator()

{

RuleFor(x => x.ProductName).NotEmpty().WithMessage("Name is required.").MaximumLength(100);

RuleFor(x => x.Quantity).GreaterThanOrEqualTo(0).WithMessage("Quantity must be non-negative.");

RuleFor(x => x.Price).GreaterThan(0).WithMessage("Price must be greater than 0.");

}

}

**For Sample, just change the Add method to incorporate FluentValidation as below:**

[HttpPost]

public async Task<IActionResult> Add([FromBody] InventoryItem item)

{

var validator = new InventoryItemValidator();

var validationResult = await validator.ValidateAsync(item);

if (!validationResult.IsValid)

{

var errors = validationResult.Errors

.Select(e => e.ErrorMessage)

.ToList();

return BadRequest(errors);

}

var addedItem = await \_service.AddAsync(item);

return CreatedAtAction(nameof(GetById), new { id = addedItem.Id },

addedItem);

}

1. Global error checking for other methods:

**Create a folder Middleware and write Global Error Handler class in the folder as below:**

using System.Net;

using System.Text.Json;

namespace InventoryService.Middleware;

public class ErrorHandlerMiddleware

{

// Reference to the next middleware in the pipeline

private readonly RequestDelegate \_next;

// Logger to log any unhandled exceptions

private readonly ILogger<ErrorHandlerMiddleware> \_logger;

// Constructor to inject dependencies: next middleware and logger

public ErrorHandlerMiddleware(RequestDelegate next, ILogger<ErrorHandlerMiddleware> logger)

{

\_next = next;

\_logger = logger;

}

// This method is called automatically for each HTTP request

public async Task Invoke(HttpContext context)

{

try

{

// Pass control to the next middleware component or the endpoint (controller)

await \_next(context);

}

catch (Exception ex)

{

// Log the exception details with a custom message

\_logger.LogError(ex, "An unhandled exception occurred");

// Set the HTTP status code to 500 (Internal Server Error)

context.Response.StatusCode = StatusCodes.Status500InternalServerError;

// Set the response content type to JSON so that the client receives a proper error object

context.Response.ContentType = "application/json";

// Create a simple error response object to return to the client

var result = new

{

message = "An unexpected error occurred.", // Generic error message

details = ex.Message // Include actual exception message (hide in production if needed)

};

// Serialize and write the error response as JSON to the response body

await context.Response.WriteAsJsonAsync(result);

}

}

}

**Line by Line Explanation of the middleware**

**🔍 Line-by-Line Explanation**

**public class ErrorHandlerMiddleware**

* This declares a middleware class named ErrorHandlerMiddleware.
* Middleware is a component that runs in the HTTP pipeline and can inspect, modify, or handle HTTP requests/responses.

**private readonly RequestDelegate \_next;**

* This represents the next middleware in the pipeline.
* Middleware components are chained, and \_next lets this component forward the request down the pipeline.

**private readonly ILogger<ErrorHandlerMiddleware> \_logger;**

* This allows you to log messages (errors, warnings, info).
* The ILogger<T> is injected automatically by ASP.NET Core’s built-in dependency injection.

**public ErrorHandlerMiddleware(RequestDelegate next, ILogger<ErrorHandlerMiddleware> logger)**

* Constructor that accepts dependencies:
  + next: the next middleware delegate
  + logger: the logging service for logging errors

**public async Task Invoke(HttpContext context)**

* This method is **mandatory** in middleware.
* It gets called for **every HTTP request**.
* HttpContext context contains request/response details like headers, status code, body, etc.

**try { await \_next(context); }**

* It forwards the request to the **next middleware or controller**.
* If no exception occurs, everything runs normally.

**catch (Exception ex)**

* If any **unhandled exception** occurs in the next middleware or controller, it will be caught here.
* This is where **global error handling** happens.

**\_logger.LogError(ex, "Unhandled exception occurred");**

* Logs the exception using the built-in logger.
* You can later view this in the console, log files, or application monitoring tools.

**context.Response.StatusCode = StatusCodes.Status500InternalServerError;**

* Sets the HTTP response status code to **500**, which means **Internal Server Error**.

**context.Response.ContentType = "application/json";**

* Ensures the response is sent back as **JSON**, so Postman/clients can parse it properly.

**var result = new { message = ..., details = ex.Message };**

* Creates an **anonymous object** for the JSON response.
* message: a generic user-facing error message.
* details: the actual exception message. (In production, you might remove this for security.)

**await context.Response.WriteAsJsonAsync(result);**

* Serializes the result object to JSON and writes it to the HTTP response body.

**Register the middle ware in Program.cs**

The complete Program.cs including the middleware is given below:

using InventoryService.Data;

using InventoryService.Interfaces;

using InventoryService.Repositories;

using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

builder.Services.AddDbContext<InventoryDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddScoped<IInventoryRepository, InventoryRepository>();

builder.Services.AddScoped<IInventoryService, InventoryService.Services.InventoryService>();

var app = builder.Build();

// Add custom middleware for error handling

app.UseMiddleware<InventoryService.Middleware.ErrorHandlerMiddleware>();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

1. Capture logs using Serilog

**To capture logs using Serilog for Console and Files, we have to install the following**:

Serilog.AspNetCore

Serilog.Sinks.Console

Serilog.Sinks.File

**Then make changes in Program.cs file**

using InventoryService.Data;

using InventoryService.Interfaces;

using InventoryService.Repositories;

using Microsoft.EntityFrameworkCore;

using Serilog;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

// Configure Serilog

Log.Logger = new LoggerConfiguration()

.WriteTo.Console()

.WriteTo.File("Logs/log.txt", rollingInterval: RollingInterval.Day)

.Enrich.FromLogContext()

.CreateLogger();

builder.Host.UseSerilog(); // Replace built-in logger with Serilog

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

builder.Services.AddDbContext<InventoryDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddScoped<IInventoryRepository, InventoryRepository>();

builder.Services.AddScoped<IInventoryService, InventoryService.Services.InventoryService>();

var app = builder.Build();

// Add custom middleware for error handling

app.UseMiddleware<InventoryService.Middleware.ErrorHandlerMiddleware>();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

1. Suppress message in Production.

In case, if we want to suppress the messages received by user in Production, we need to change our Custom Error Handler as below:

using System.Net;

using System.Text.Json;

namespace InventoryService.Middleware;

public class ErrorHandlerMiddleware

{

// Reference to the next middleware in the pipeline

private readonly RequestDelegate \_next;

// Logger to log any unhandled exceptions

private readonly ILogger<ErrorHandlerMiddleware> \_logger;

private readonly IWebHostEnvironment \_env;

// Constructor to inject dependencies: next middleware and logger

public ErrorHandlerMiddleware(RequestDelegate next, ILogger<ErrorHandlerMiddleware> logger, IWebHostEnvironment env)

{

\_next = next;

\_logger = logger;

\_env = env;

}

// This method is called automatically for each HTTP request

public async Task Invoke(HttpContext context)

{

try

{

// Pass control to the next middleware component or the endpoint (controller)

await \_next(context);

}

catch (Exception ex)

{

// Log the exception details with a custom message

\_logger.LogError(ex, "An unhandled exception occurred");

// Set the HTTP status code to 500 (Internal Server Error)

context.Response.StatusCode = StatusCodes.Status500InternalServerError;

// Set the response content type to JSON so that the client receives a proper error object

context.Response.ContentType = "application/json";

// Create a simple error response object to return to the client

//var result = new

//{

// message = "An unexpected error occurred.", // Generic error message

// details = ex.Message // Include actual exception message (hide in production if needed)

//};

var result = \_env.IsDevelopment()

? new

{

message = "An unexpected error occurred.",

details = ex.Message // Show detailed error in development

}

: new

{

message = "An unexpected error occurred. Please contact support.", // No internal details in production

details = string.Empty

};

// Serialize and write the error response as JSON to the response body

await context.Response.WriteAsJsonAsync(result);

}

}

}