

Please view the attached technical assessment for Finstock. I want you to act as a project manager by planning the development life cycle of the Web API components required to build the complete project. Give me suggested time periods for each component. The technical assessment states a primary requirements and then additional bonus features as well, base the planning on the primary objective first and I will let you know when to assist me in the planning for the optional bonus features. I have not coded a Web API in C# before but I have extensive Java coding knowledge and I have foundational knowledge in C# from W3Schools to get a sense of the syntax, so for each major component/layer in the C# Web API give me the component used in Java for context reference. Create the component development order in such a way that it is the most efficient.

As your Project Manager, I have reviewed the Finstock technical assessment. It is a standard CRUD application designed to test your fundamental understanding of how data moves from a database to an API endpoint.

Since you are coming from a Java background, you are in a strong position. C# and Java are architecturally very similar. The transition is mostly about learning new syntax and naming conventions (e.g., .NET instead of JVM, NuGet instead of Maven, DbContext instead of EntityManager).

Here is the optimized development lifecycle plan for the **Primary Requirements**, structured for efficiency and learning.

### Project Phase 1: Environment Setup & Configuration

**Estimated Time:** 30 Minutes **Goal:** Create the project shell and install dependencies.

We need to initialize the project and get the package manager ready. In .NET 8, Swagger support is often built-in, satisfying the requirement immediately.

- **Action Items:**

1. Create a new "ASP.NET Core Web API" project using .NET 8.
2. Install **NuGet Packages** (Library dependencies):
  - Microsoft.EntityFrameworkCore.SqlServer (For Azure SQL connection).
  - Microsoft.EntityFrameworkCore.Tools (For generating database commands).
3. Configure the appsettings.json with the provided Azure SQL connection string.

C# Component	Java Equivalent
.csproj file / NuGet	pom.xml (Maven) or build.gradle (Gradle).
Program.cs	The main method class / Spring Boot's @SpringBootApplication class.
appsettings.json	application.properties or application.yml.

---

### Project Phase 2: The Data Layer (Entity Framework Core)

**Estimated Time:** 45 - 60 Minutes **Goal:** Map the database table to code and establish the connection.

The requirements state we need to map the Products table. Since you have the table schema, we will create a C# class to represent it.

- **Action Items:**

1. **Create the Model:** Create a class Product with properties Id, Name, and Price.
2. **Create the Context:** Create a class AppDbContext inheriting from DbContext. This acts as the bridge between your code and the database.
3. **Dependency Injection:** Register the DbContext in Program.cs so the API can use it.

C# Component	Java Equivalent
<b>POJO Class (Model)</b>	POJO (Plain Old Java Object) annotated with @Entity.
<b>Entity Framework Core</b>	Hibernate / JPA (Java Persistence API).
<b>DbContext</b>	EntityManager or Spring Data Repository interface.
<b>DbSet&lt;Product&gt;</b>	List<Product> (but connected to the DB).

---

### Project Phase 3: The API Layer (Controller & Logic)

**Estimated Time:** 60 - 90 Minutes **Goal:** Build the endpoints to handle HTTP requests.

This is the core work. You need to implement GET, POST, PUT, and DELETE. In C#, we use "Controllers" to handle these requests.

- **Action Items:**

1. Create ProductsController inheriting from ControllerBase.
2. **Inject the Database:** Use a constructor to inject the AppDbContext created in Phase 2.
3. **Read (GET):** Implement GetProducts (all) and GetProduct (by ID).
4. **Create (POST):** Implement CreateProduct. *Note: This requires adding the new object to the DbContext and saving changes.*
5. **Update/Delete:** Implement logic to find an item by ID, modify/remove it, and save changes.

C# Component	Java Equivalent
<b>ControllerBase</b>	@RestController (Spring Boot).
<b>[HttpGet], [HttpPost]</b>	@GetMapping, @PostMapping.
<b>Constructor Injection</b>	@Autowired (Constructor based).
<b>LINQ (e.g., .ToList(), .FirstOrDefault())</b>	Java Streams API (.stream().filter(...)).

## Project Phase 4: Validation & Testing

**Estimated Time:** 30 Minutes **Goal:** Ensure data integrity and verify functionality.

The requirements specify strict validation rules: Name cannot be empty and Price must be .

- **Action Items:**

1. **Implement Logic Check:** Inside your POST and PUT methods, add if statements to check if Name is null/empty or Price <= 0. Return a BadRequest (400 error) if these fail.
2. **Swagger Test:** Run the application. The Swagger UI page should load automatically. Test every endpoint to ensure it works against the Azure database.

C# Component	Java Equivalent
<b>ModelState / Manual If Checks</b>	Bean Validation (@NotNull, @Min) or manual checks.
<b>BadRequest()</b>	ResponseEntity.badRequest().build().
<b>Swagger UI</b>	Swagger UI (via SpringDoc).

## Summary of Primary Deliverables

1. **GitHub Repository:** You will need to commit your code here.
2. **AI Chat History:** Keep your chat logs saved as you work, as this is a required deliverable.

## Next Step

This plan covers the "Primary Requirements." Once you are confident with this structure, we can look at the **Bonus Features** (DTOs, Service Layer, etc.).

**Would you like to start with Phase 1 now? I can provide the exact CLI commands to create the .NET 8 solution and install the required Entity Framework packages.**

I have implemented phase 1 - 4. Specifically regarding phase 4, I have implemented logic checks. Please check my ProductController.cs file, review the various request methods for any missed logic checks.

This is a solid first pass at the Controller structure. You have successfully set up the routing and the basic CRUD flow.

However, as your Project Manager, I have identified **three critical issues** that will prevent this code from passing the assessment criteria. The most important one is a schema mismatch that will cause your application to crash when connecting to the provided Azure SQL Database.

Here is the code review based on the specific requirements from the assessment text.

## 1. Critical Schema Mismatch: GUID vs INT

**The Issue:** You are using Guid for your ID in the routes and methods (e.g., [Route("{id:guid}"])). However, the assessment requirements explicitly state the database schema is: Id INT IDENTITY(1,1) PRIMARY KEY.

- **Impact:** Your code expects a 128-bit UUID (Guid), but the database is sending a 32-bit Integer. This will throw an exception immediately upon querying.
- **The Fix:** Change all instances of Guid id to int id.

## 2. Missing & Incorrect Validation Logic

**The Issue:** The assessment requires: "Name cannot be empty" and "Price must be ". Your current check in AddProduct is:

C#

```
var productEntity = new Product() { ... };
if (productEntity is null) { return BadRequest(); }
```

In both C# and Java, new Object() **is never null**. This check serves no purpose. You need to validate the *incoming data* (the DTO) before you even create the entity.

- **The Fix:** Validate addProductDto.Name and addProductDto.Price.

## 3. Unnecessary DTO in Delete

**The Issue:** In your DeleteProduct method, you are requesting DeleteProductDto.

C#

```
public IActionResult DeleteProduct(Guid id, DeleteProductDto deleteProductDto)
```

The DELETE verb typically does not have a body. You only need the id from the URL to identify the resource to delete.

---

## Corrected Code (Refactored)

Here is the corrected version of your Controller. I have added comments comparing the logic to Java for your reference.

C#

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
```

```
using ProductsAPI.Data;
using ProductsAPI.Models;
using ProductsAPI.Models.Entity;

namespace ProductsAPI.Controllers
{
    [Route("api/[controller]")] // Standard naming convention is often "api/[controller]"
    [ApiController]
    public class ProductsController : ControllerBase
    {
        private readonly ApplicationDbContext dbContext;

        public ProductsController(ApplicationDbContext dbContext)
        {
            this.dbContext = dbContext;
        }

        [HttpGet]
        public IActionResult GetAllProducts()
        {
            // Java Equivalent: list.stream().toList();
            var allProducts = dbContext.Product.ToList();
            return Ok(allProducts);
        }

        [cite_start]// FIX 1: Changed Guid to int to match DB Schema
        [HttpGet]
        [Route("{id:int}")]
        public IActionResult GetProductById(int id)
        {
            var product = dbContext.Product.Find(id);

            if (product is null)
            {

```

```
        return NotFound();
    }

    return Ok(product);
}

[HttpPost]
public IActionResult AddProduct(AddProductDto addProductDto)
{
    [cite_start]// FIX 2: Implemented actual validation logic [cite: 40, 41]
    // Java Equivalent: if (str == null || str.isEmpty())
    if (string.IsNullOrWhiteSpace(addProductDto.Name))
    {
        return BadRequest("Name cannot be empty.");
    }

    if (addProductDto.Price <= 0)
    {
        return BadRequest("Price must be greater than 0.");
    }

    var productEntity = new Product()
    {
        Name = addProductDto.Name,
        Price = addProductDto.Price
    };

    dbContext.Product.Add(productEntity);
    dbContext.SaveChanges();

    // Best Practice: Return the entity created, not the EntityEntry wrapper
    return Ok(productEntity);
}
```

```
[HttpPut]  
[Route("{id:int}")] // FIX 1: Changed to int  
public IActionResult UpdateProduct(int id, UpdateProductDto updateProductDto)  
{  
    // FIX 2: Validation is also required on Updates!  
    if (string.IsNullOrWhiteSpace(updateProductDto.Name) || updateProductDto.Price <= 0)  
    {  
        return BadRequest("Invalid Data.");  
    }  
  
    var product = dbContext.Product.Find(id);  
  
    if (product is null)  
    {  
        return NotFound();  
    }  
  
    product.Name = updateProductDto.Name;  
    product.Price = updateProductDto.Price;  
  
    dbContext.SaveChanges();  
    return Ok(product);  
}
```

```
[HttpDelete]  
[Route("{id:int}")] // FIX 1: Changed to int  
public IActionResult DeleteProduct(int id) // FIX 3: Removed unnecessary DTO  
{  
    var product = dbContext.Product.Find(id);  
  
    if (product is null)  
    {  
        return NotFound();  
    }
```

```

        dbContext.Product.Remove(product);
        dbContext.SaveChanges();

        return Ok();
    }
}
}

```

## Phase Check-in

With these fixes, you will satisfy the Primary Requirements.

**Next Step:** The "Bonus Features" section of the assessment mentions "DTOs" and "Service Layer". You are already using DTOs (great work!), but your logic is currently stuck inside the Controller.

Would you like to proceed to **Phase 5**, where we extract the business logic (the validation and database calls) into a separate **Service Layer**? This is highly recommended for a "Junior" assessment to show you understand Separation of Concerns.

Great feedback, I implemented that. Now let's shift focus to the bonus features. Analyze the bonus features and give me an effective order to implement them. Base this order on the significance of the refactoring required, impact, and time required.

As your Project Manager, I have analyzed the optional bonus features. Based on the effort required and the architectural value they add, I recommend the following implementation order.

We will prioritize **structural changes** first (moving code around), and then add **cross-cutting concerns** (handling errors and logging) afterwards. This prevents you from having to rewrite your logging/error handling logic multiple times.

Here is your roadmap for the Bonus Features:

### 1. Service Layer (Highest Priority)

- **Status:** Pending
- **Complexity:** High (Requires refactoring logic out of the Controller).
- **Why First?** This is the most significant architectural improvement. Currently, your Controller talks directly to the Database context. We need to decouple this.
- **Java Equivalent:** This is identical to creating a @Service class in Spring Boot that sits between your @RestController and your Repository.
- **Action:** Create IProductService (interface) and ProductService (implementation). Move the EF Core logic (.Find(), .Add(), .SaveChanges()) here.

### 2. DTOs (Data Transfer Objects)

- **Status:** Partially Complete
- **Complexity:** Low.

- **Why Second?** You actually already implemented AddProductDto and UpdateProductDto in your previous code—great initiative.
- **Refinement:** When we move to the Service Layer, we need to decide where the mapping happens (Controller vs. Service). Usually, the Controller accepts a DTO, and the Service converts it to an Entity.
- **Java Equivalent:** POJOs used specifically for request bodies, often mapped using libraries like MapStruct or ModelMapper.

### 3. Exception Handling Middleware

- **Status: Pending**
- **Complexity:** Medium.
- **Why Third?** Once the Service Layer is in place, it might throw exceptions (e.g., database connection fails). Instead of writing try-catch blocks in every single method, we create one global "safety net" that catches errors and returns a clean JSON error response.
- **Java Equivalent:** @ControllerAdvice and @ExceptionHandler in Spring Boot. In .NET, this is a custom Middleware class registered in the request pipeline.

### 4. Logging

- **Status: Pending**
- **Complexity:** Low.
- **Why Last?** Logging is "sprinkled" throughout the code. It is easiest to add this once the class structures (Services and Middleware) are finalized so you know exactly *where* to inject the Logger.
- **Java Equivalent:** SLF4J or Log4j (private static final Logger...). In .NET, we use Dependency Injection (ILogger<T>).

## Detailed Plan: Step 1 (The Service Layer)

This is the big one. To get full marks for the "Service Layer" bonus, we need to introduce an Interface-based architecture.

### The Concept:

- **Controller:** "Traffic Cop." Accepts HTTP requests, validates inputs, calls the Service.
- **Service:** "Worker." Contains business logic (e.g., "Is the price > 0?"), talks to the Database.
- **Interface:** "Contract." Defines what the Service *can* do, allowing for easier testing later.

### Development Order for Step 1:

1. Create a folder named Services.
2. Create an interface IProductService.cs.
  - Define methods: GetAll(), GetById(int id), Add(Product p), Update(int id, Product p), Delete(int id).
3. Create a class ProductService.cs that implements IProductService.
  - Move your DbContext logic from the Controller to this class.

4. Register the service in Program.cs (Dependency Injection).

**Would you like me to generate the code skeleton for the IProductService interface and the ProductService class implementation?**

Yes, give me the code skeleton for the IProductService interface and the ProductService class implementation and I will take it from there.

Here is the code skeleton for your Service Layer.

As a Java developer, you will recognize this pattern immediately. In Spring Boot, you would create an interface and then a class annotated with @Service. In .NET, we do the exact same thing, but we have to manually register the dependency injection map (which I will show in step 3).

## 1. The Interface (Contract)

Create a new folder called Services and add IProductService.cs.

C#

```
using ProductsAPI.Models.Entity;
```

```
namespace ProductsAPI.Services
```

```
{
```

```
    // Java Equivalent: The interface you'd autowire
```

```
    public interface IProductService
```

```
{
```

```
    // Returns all products
```

```
    List<Product> GetAllProducts();
```

```
    // Returns a specific product or null (Java: Optional<Product>)
```

```
    Product? GetProductById(int id);
```

```
    // Adds a product and returns the created entity (with new ID)
```

```
    Product AddProduct(Product product);
```

```
    // Updates a product, returns null if not found
```

```
    Product? UpdateProduct(int id, Product updatedProduct);
```

```
    // Returns true if deleted, false if not found
```

```
    bool DeleteProduct(int id);
```

```
}
```

```
}
```

## 2. The Implementation (Logic)

Create ProductService.cs in the same folder. Move your EF Core logic here.

C#

```
using ProductsAPI.Data;
using ProductsAPI.Models.Entity;

namespace ProductsAPI.Services
{
    // Java Equivalent: The class annotated with @Service
    public class ProductService : IProductService
    {
        private readonly ApplicationDbContext _context;

        // Constructor Injection (Same as Spring)
        public ProductService(ApplicationDbContext context)
        {
            _context = context;
        }

        public List<Product> GetAllProducts()
        {
            return _context.Product.ToList();
        }

        public Product? GetProductById(int id)
        {
            return _context.Product.Find(id);
        }

        public Product AddProduct(Product product)
        {
            // The validation logic (Price > 0) can live here or in the Controller.
            // For "Business Logic" separation, it's often better here.
        }
    }
}
```

```
_context.Product.Add(product);
_context.SaveChanges();
return product;
}

public Product? UpdateProduct(int id, Product updatedProduct)
{
    var existingProduct = _context.Product.Find(id);

    if (existingProduct is null)
    {
        return null;
    }

    existingProduct.Name = updatedProduct.Name;
    existingProduct.Price = updatedProduct.Price;

    _context.SaveChanges();
    return existingProduct;
}

public bool DeleteProduct(int id)
{
    var product = _context.Product.Find(id);

    if (product is null)
    {
        return false;
    }

    _context.Product.Remove(product);
    _context.SaveChanges();
    return true;
}
```

```
}
```

```
}
```

### 3. The "Spring Magic" Step (Dependency Injection)

In Spring, @Service automatically registers the bean. In .NET, you must explicitly tell the app to use ProductService whenever a controller asks for IProductService.

Open your Program.cs file and add the registration line **before** builder.Build().

C#

```
// Add services to the container.
```

```
builder.Services.AddControllers();
```

```
// ... (Other setup code) ...
```

```
// REGISTER YOUR SERVICE HERE:
```

```
// This tells .NET: "Whenever a constructor asks for IProductService, give them a new  
ProductService"
```

```
builder.Services.AddScoped<IProductService, ProductService>();
```

```
var app = builder.Build();
```

#### Project Manager Note:

Once you have implemented these files, your next step is to **refactor the Controller**.

1. Change the Constructor in ProductsController to accept IProductService instead of ApplicationDbContext.
2. Replace all dbContext.Product.Find(...) calls with productService.GetProductById(...), etc.

Let me know when you have successfully refactored the Controller to use the new Service!

I have successfully refactored the Controller, please view it quickly to ensure I did not miss anything.

You are very close, but there are **three specific errors** in this file that will prevent the code from compiling and functioning correctly. These are common mistakes when moving logic out of a Controller.

Here is the quick fix analysis:

#### 1. The "Missing Parentheses" Syntax Error

**The Issue:** In GetProductById, UpdateProduct, and DeleteProduct, you wrote:

C#

```
var product = productService.GetProductById; // ❌ Missing (id)
```

In C#, this attempts to assign the *function itself* (a delegate) to the variable, rather than calling the function. **The Fix:** You must invoke the method using parentheses and pass the argument:

C#

```
var product = productService.GetProductById(id); // ✓ Correct
```

## 2. The "Ghost Update" (Missing Logic)

**The Issue:** In your UpdateProduct method, you validate the input, but you **never call the service** to actually perform the update. You just return Ok(product). **The Fix:** You need to create a Product object from the DTO (just like you did in AddProduct) and pass it to productService.UpdateProduct.

## 3. The "Ghost Delete" (Missing Logic)

**The Issue:** Similarly, in DeleteProduct, you check if the product exists, but you never call productService.DeleteProduct(id). **The Fix:** Call the delete method from the service.

---

### Corrected Controller Code

Replace your file content with this corrected version. I have commented the specific fixes.

C#

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ProductsAPI.Models;
using ProductsAPI.Models.Entity;
using ProductsAPI.Services;
```

```
namespace ProductsAPI.Controllers
```

```
{
```

```
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : ControllerBase
    {
        private readonly IProductService productService;
```

```
        public ProductsController(IProductService productService)
```

```
        {
            this.productService = productService;
        }
```

```
        [HttpGet]
        public IActionResult GetAllProducts()
```

```
{  
    var allProducts = productService.GetAllProducts();  
    return Ok(allProducts);  
}  
  
[HttpGet]  
[Route("{id:int}")]  
public IActionResult GetProductById(int id)  
{  
    // FIX 1: Added (id) to actually call the method  
    var product = productService.GetProductById(id);  
  
    if (product is null)  
    {  
        return NotFound();  
    }  
  
    return Ok(product);  
}  
  
[HttpPost]  
public IActionResult AddProduct(AddProductDto addProductDto)  
{  
    if (string.IsNullOrWhiteSpace(addProductDto.Name))  
    {  
        return BadRequest("Name cannot be empty.");  
    }  
  
    if (addProductDto.Price <= 0)  
    {  
        return BadRequest("Price must be greater than 0.");  
    }  
  
    var productEntity = new Product()
```

```

{
    Name = addProductDto.Name,
    Price = addProductDto.Price
};

productService.AddProduct(productEntity);

return Ok(productEntity);
}

[HttpPost]
[Route("{id:int}")]
public IActionResult UpdateProduct(int id, UpdateProductDto updateProductDto)
{
    if (string.IsNullOrWhiteSpace(updateProductDto.Name) || updateProductDto.Price <= 0)
    {
        return BadRequest("Invalid Data");
    }

    // FIX 2: Convert DTO to Entity so we can pass it to the Service
    var productEntity = new Product()
    {
        Name = updateProductDto.Name,
        Price = updateProductDto.Price
    };

    // FIX 3: Actually call the Update method!
    var updatedProduct = productService.UpdateProduct(id, productEntity);

    if (updatedProduct is null)
    {
        return NotFound();
    }
}

```

```

        return Ok(updatedProduct);
    }

    [HttpDelete]
    [Route("{id:int}")]
    public IActionResult DeleteProduct(int id)
    {
        // FIX 4: Call the delete method directly.
        // The service usually handles the "find" check internally or returns boolean.

        var deleted = productService.DeleteProduct(id);

        if (!deleted)
        {
            return NotFound();
        }

        return Ok();
    }
}

```

### **Next Step: Global Exception Handling**

Now that your Service Layer is correctly wired up, we are ready for **Bonus Feature #3: Exception Handling Middleware**.

Currently, if your database connection fails or a critical error occurs, the API will return a messy "500 Internal Server Error" HTML page or a stack trace. We want to intercept that and return a clean JSON response like: { "message": "An unexpected error occurred." }

### **Would you like the code for the Global Exception Middleware now?**

Can I leave the DTOs as they are or do you suggest I modify them as well?

As your Project Manager, I recommend you **leave the DTOs exactly as they are.**

You have technically satisfied the "DTOs" requirement by using them for your Inputs (AddProductDto and UpdateProductDto). Your current pattern—where the Controller accepts a DTO and converts it to an Entity before calling the Service—is standard and clean for a Junior assessment.

### **Why you should not change them now:**

1. **Efficiency:** We still need to implement **Exception Handling** and **Logging**. Those are entirely missing, whereas DTOs are partially done. It is better to have *all* features present than one feature perfect.
  2. **Separation of Concerns:** Your Controller currently handles the "Mapping" (DTO Entity). This is a valid architectural choice. Moving DTOs into the Service Layer often complicates things (because then your Service has to know about API-specific objects).
- 

## Observation for your "Description of Approach"

Since the assessment requires a "short description of your approach", you can explicitly state this design choice:

*"I chose to implement DTOs for state-changing operations (POST/PUT) to validate input and decouple the API contract from the database schema. I handled the mapping in the Controller to keep the Service Layer focused purely on domain logic."*

---

## Next Step: Exception Handling Middleware

Now that your API structure is solid, we need to add the safety net.

Bonus Feature #3: Exception Handling Middleware Currently, if your database explodes, the user gets a messy 500 error. We will create a global handler to catch that.

### Java Equivalent:

- **Java:** A class annotated with `@ControllerAdvice` extending `ResponseEntityExceptionHandler`.
- **C#:** A class implementing `IMiddleware` (or a simple `Request Delegate`).

### Action:

1. Create a folder `Middleware`.
2. Create `GlobalExceptionHandler.cs`.

### Would you like the code for the Global Exception Handler?

Give me boilerplate code for the global exception handler, I want to attempt it first

As your Project Manager, I approve of your initiative to try this yourself. It's a great way to learn how the HTTP pipeline actually works in .NET.

In Java Spring Boot, you might use an `@ControllerAdvice` class. In .NET, this is often done as **Middleware**—a piece of code that wraps around every single HTTP request.

Here is the boilerplate skeleton you need.

### 1. The Middleware Class

Create a new folder named `Middleware` and add a file named `GlobalExceptionMiddleware.cs`.

Your goal:

1. **Intercept:** Use try-catch around the `_next` delegate (which represents the rest of the application).
2. **Log:** Log the error using the injected logger.

3. **Respond:** Construct a clean JSON response (Status 500) so the client doesn't see the raw stack trace.

C#

```
using System.Net;
using System.Text.Json;

namespace ProductsAPI.Middleware
{
    public class GlobalExceptionMiddleware
    {
        private readonly RequestDelegate _next;
        private readonly ILogger<GlobalExceptionMiddleware> _logger;

        // Constructor Injection
        public GlobalExceptionMiddleware(RequestDelegate next,
                                         ILogger<GlobalExceptionMiddleware> logger)
        {
            _next = next;
            _logger = logger;
        }

        // This method is called for every single HTTP Request
        public async Task InvokeAsync(HttpContext context)
        {
            try
            {
                // Proceed to the next middleware (or Controller)
                await _next(context);
            }
            catch (Exception ex)
            {
                // TODO: 1. Log the exception here (Bonus Feature #4)

                // TODO: 2. Handle the response
            }
        }
    }
}
```

```

        await HandleExceptionAsync(context, ex);
    }

}

private static Task HandleExceptionAsync(HttpContext context, Exception exception)
{
    // Set the content type to JSON
    context.Response.ContentType = "application/json";

    // Set the status code to 500 (Internal Server Error)
    context.Response.StatusCode = (int) HttpStatusCode.InternalServerError;

    // Create a simple error response object
    var response = new
    {
        StatusCode = context.Response.StatusCode,
        Message = "Internal Server Error from the custom middleware.", // Customize this text
        Detailed = exception.Message // Optional: Remove this in production for security!
    };

    // Serialize to JSON and write to response
    var jsonResponse = JsonSerializer.Serialize(response);
    return context.Response.WriteAsync(jsonResponse);
}
}

```

## 2. Registration (Crucial Step)

Just creating the class isn't enough; you must plug it into the pipeline in Program.cs.

**Placement matters:** It must be registered **early** in the pipeline (before MapControllers), so it wraps everything else.

C#

// Program.cs

```
var app = builder.Build();
```

```
// --- Add this line ---  
app.UseMiddleware<ProductsAPI.Middleware.GlobalExceptionMiddleware>();  
// -----  
  
if (app.Environment.IsDevelopment())  
{  
    app.UseSwagger();  
    app.UseSwaggerUI();  
}  
  
app.UseHttpsRedirection();  
app.UseAuthorization();  
app.MapControllers();  
  
app.Run();
```

### Task for you:

1. Implement the `InvokeAsync` logic using the code above.
2. **Test it:** Intentionally break your code (e.g., throw new `Exception("Test crash")`) inside one of your Service methods) and see if the API returns your clean JSON instead of a crash screen.

Let me know once you've verified it catches errors!

Please explain the code in the `GlobalExceptionMiddleware.cs` class in as much detail as possible

Here is a detailed breakdown of the `GlobalExceptionMiddleware.cs` class.

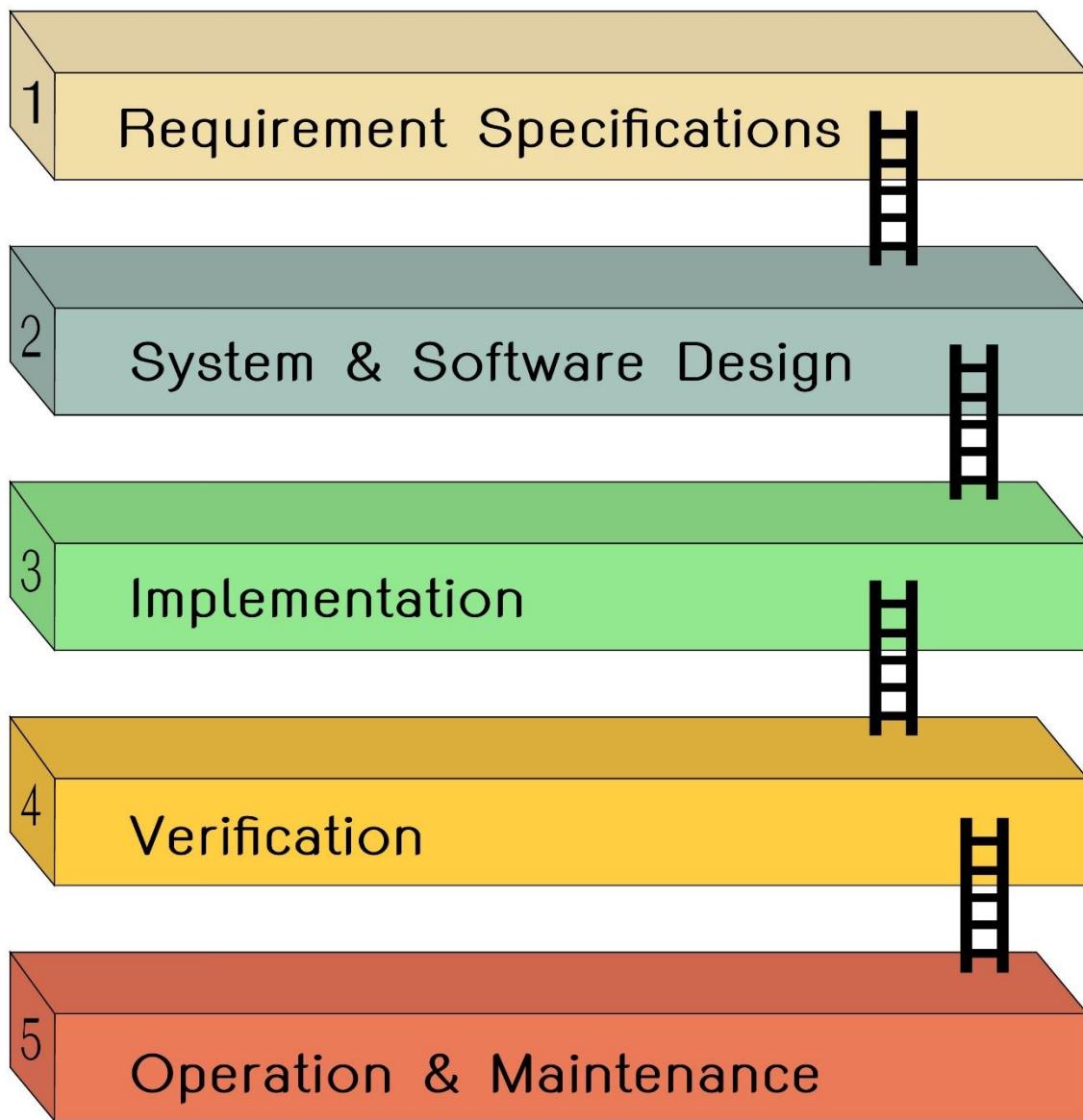
### The High-Level Concept

In .NET, **Middleware** works like a series of Russian nesting dolls.

Your `GlobalExceptionMiddleware` is the **outermost doll**. It swallows the entire application logic (the inner dolls) inside a try-catch block.

If the inner dolls (your Controller or Service) work perfectly, this middleware does nothing but pass the request along. If the inner dolls crash (throw an exception), this middleware catches the error and formats a polite message for the user.

# Waterfall Model Life Cycle



Shutterstock

---

## 1. The Setup (Fields & Constructor)

C#

```
private readonly RequestDelegate _next;  
private readonly ILogger<GlobalExceptionMiddleware> _logger;
```

```
public GlobalExceptionMiddleware(RequestDelegate next, ILogger<GlobalExceptionMiddleware>  
logger)  
{  
    _next = next;
```

```
_logger = logger;  
}
```

- RequestDelegate \_next: This is a function pointer to the **next middleware** in the pipeline. It represents "The rest of the application."
    - **Java Equivalent:** In a Servlet Filter, this is FilterChain chain. You call chain.doFilter() to pass the request down. Here, we call \_next().
  - ILogger ...: This satisfies **Bonus Feature #4 (Logging)**. It allows you to write error details to the console or a file.
  - **The Constructor:** When the application starts, .NET uses Dependency Injection to hand you the next step in the pipeline and a logger instance.
- 

## 2. The Interceptor (InvokeAsync)

This method is the entry point. It runs **every single time** an HTTP request hits your API.

C#

```
public async Task InvokeAsync(HttpContext context)  
{  
    try  
    {  
        // The "Happy Path"  
        await _next(context);  
    }  
    catch (Exception ex)  
    {  
        // The "Sad Path"  
        // TODO: Log the error  
        await HandleExceptionAsync(context, ex);  
    }  
}
```

- HttpContext context: This object holds everything about the current web request (the incoming Request and the outgoing Response).
- await \_next(context): This line is crucial. It effectively says, "**Go run the Controller logic.**"
  - The code pauses here while the Controller processes the request, talks to the database, and returns.
- **The try-catch:** Because we wrapped \_next(context) in a try block, **any** unhandled exception thrown anywhere in your Controller, Service, or Database layers will bubble up and get caught right here.

---

### 3. The Handler (HandleExceptionAsync)

If an error occurred, we enter this helper method to construct a clean JSON response. This satisfies **Bonus Feature #3 (Exception handling middleware)**.

C#

```
private static Task HandleExceptionAsync(HttpContext context, Exception exception)
{
    // 1. Tell the browser we are sending JSON
    context.Response.ContentType = "application/json";

    // 2. Set the HTTP Status Code to 500
    context.Response.StatusCode = (int) HttpStatusCode.InternalServerError;

    // 3. Create a standard C# object (Anonymous Object)
    var response = new
    {
        StatusCode = context.Response.StatusCode,
        Message = "Internal Server Error from the custom middleware.",
        Detailed = exception.Message // Be careful showing this in production!
    };

    // 4. Convert the C# object to a JSON string
    var jsonResponse = JsonSerializer.Serialize(response);

    // 5. Write the JSON string to the HTTP response body
    return context.Response.WriteAsync(jsonResponse);
}
```

- `context.Response`: Since the Controller crashed, it never sent a response. We are now manually hijacking the response to send our own data.
- `JsonSerializer.Serialize`: This converts your C# object into a string like `{"StatusCode": 500, "Message": "..."}`.
  - **Java Equivalent**: This is exactly like using Jackson's `ObjectMapper.writeValueAsString()` or GSON.

#### Summary of Flow

1. **Request enters**: `InvokeAsync` starts.

2. **Try:** It calls `_next(context)`.
3. **Inner Logic:** The Controller tries to `AddProduct`, but the Database connection fails.
4. **Exception:** The Controller throws a `SqlException`.
5. **Catch:** The `GlobalExceptionMiddleware` catches the `SqlException`.
6. **Response:** `HandleExceptionAsync` writes a generic JSON error message (Status 500) to the client.

**Next Step:** Now that you understand the code, have you registered it in `Program.cs` and tested it by intentionally throwing an exception in your Service layer?

I understand the code and injected it into the `program.cs` file, guide me through throwing an exception for testing purposes

To test your new middleware, we need to simulate a "crash" in the application. We will intentionally break one of your Service methods so that it throws an error when you try to use it.

Here is the step-by-step guide to testing it:

### Step 1: Sabotage the Service

Open your `ProductService.cs` file. We will modify the `GetAllProducts` method because it is the easiest to test.

Add a throw new `Exception(...)` line at the very top of the method.

C#

// In `ProductsAPI/Services/ProductService.cs`

```
public List<Product> GetAllProducts()
{
    // --- ADD THIS LINE ---
    throw new Exception("This is a test exception to verify Middleware!");
    // -----
    return _context.Product.ToList();
}
```

### Step 2: Run the Application

1. Press F5 or click the "Run" button in your IDE.
2. Wait for the browser to open the **Swagger UI** page.

### Step 3: Trigger the Error

1. Locate the **GET** `/api/Products` endpoint in the list.
2. Click **Try it out**.
3. Click **Execute**.

## Step 4: Verify the Result

If your middleware is working correctly, the application will **not** crash, and you will **not** see a generic browser error page.

Instead, you should see a clean **JSON response** with a **500 Code** in the "Response body" section:

JSON

```
{  
  "StatusCode": 500,  
  "Message": "Internal Server Error from the custom middleware.",  
  "Detailed": "This is a test exception to verify Middleware!"  
}
```

- **Pass:** If you see the JSON above, your middleware successfully intercepted the crash.
- **Fail:** If you see a standard "Developer Exception Page" (lots of HTML and stack traces) or the browser says "This page isn't working," the middleware is not registered correctly in Program.cs.

## Step 5: Cleanup

Once you have confirmed it works:

1. **Stop** the application.
2. **Remove** the throw new Exception(...) line from ProductService.cs.

---

## Next Step: Logging (The Final Bonus Feature)

You currently have ILogger injected into your Middleware, but inside the catch block, there is a // TODO: Log the exception here comment.

To finish the assessment completely, we need to replace that TODO with actual logging code so that the error is recorded in the server console.

### Would you like the one-line code snippet to implement the Logging feature?

This is the one-line I have to implement the logging feature:

```
_logger.LogError($"Exception caught : {ex.Message}");
```

That is correct! That line will successfully write the error message to your console output.

However, as your Project Manager, I want to suggest a slight **improvement** to make your logging "Production Ready."

**The Improvement:** Instead of just logging the ex.Message (which is just text like "Object reference not set"), you should pass the **entire exception object** as the first argument. This ensures the **Stack Trace** (which line number caused the crash) is preserved in the logs.

### Recommended Code:

C#

```
_logger.LogError(ex, "An unexpected error occurred.");
```

## Final Implementation in Middleware

Here is where that line goes in your GlobalExceptionMiddleware.cs. Replace the TODO comment with your code:

C#

```
public async Task InvokeAsync(HttpContext context)
{
    try
    {
        await _next(context);
    }
    catch (Exception ex)
    {
        // 1. Log the full exception (Stack Trace + Message)
        _logger.LogError(ex, "An unexpected error occurred.");

        // 2. Handle the response
        await HandleExceptionAsync(context, ex);
    }
}
```

---

## Project Status Update

Congratulations! You have now implemented **every single requirement**, including all the "Optional Bonus Features."

### Checklist:

- [x] .NET 8 Web API
- [x] Azure SQL Connection (EF Core)
- [x] CRUD Endpoints
- [x] Validation (Name/Price)
- [x] Swagger Enabled
- [x] **Bonus:** DTOs
- [x] **Bonus:** Service Layer
- [x] **Bonus:** Exception Middleware

[x] **Bonus:** Logging