TECHNICAL SPECIFICATION DOCUMENT

SHGARDI BACKEND TASK
Final Round

Jaeson Karter Joseph
11/02/2025

Problem Statement

Final Round Technical project (Backend) Inbox ×

Prem Shgardi Sun, Feb 9, 2:19 AM (2 days ago)
to me

Hello there,

Thank you very much for investing your time and effort to interview with our team about the Backend Developer position at SHGARDI.
All of us really enjoyed meeting you.

Kindly find the task below

Overview:

* Create a simple REST API (using ASP.NET Core) that manages a small set of entities (e.g., "Products" with properties like name, description, price). The API should support basic CRUD operations (Create, Read, Update, Delete) and use Entity Framework Core for data access.
Notes:

* Try to implement as much as you can, even if you didn't finish the essential features your submission will be accepted you'll just have a lower evaluation.
* We are keen on simplicity, easy-to-read code, documentation, and bug-free code.
* Your project should have a git repository, and your git history will be considered in the evaluation of the task (The steps of your solution).
* Your code will be evaluated on how clean and modular it is.

Introduction

Project Repo: https://github.com/jaeson-karter-joseph/ShgEcom_Backend_Task

This document serves as the **technical blueprint** for the **Shgardi Backend Task**, outlining the architecture, design principles, technologies, and implementation details of the system. It provides a comprehensive overview of the backend development process, ensuring scalability, maintainability, and performance optimization.

The primary goal of this backend system is to **deliver a high-performance, secure, and scalable API** that supports core business operations. It follows **Clean Architecture**, **CQRS**, and **Microservices Best Practices** while leveraging **.NET Core** and **MongoDB** to ensure efficient data management.

Through this document, we aim to provide a clear roadmap for developers, stakeholders, and decision-makers to understand the underlying technical structure and operational flow of the application.

Technologies Used

- ✅ **Technology** Stack: .NET Core 8, MongoDB (Cloud), CQRS, Mediator, JWT Authentication
- ✅ **Architecture**: Clean Architecture with Domain-Driven Design (DDD)
- ✅ **Scalability**: Built for high-traffic workloads with optimized database queries
- ✅ **Security**: Implements JWT Authentication
- ✅ **Error Boundary**: Global Error Handling
- ✅ **API Standards**: RESTful API with Swagger Documentation
- ✅ **File Logger**: Global Logger Middleware to Log All Request & Response
- ✅ **Cache Implementation**: In Memory Cache Implemented with Eviction Policy for faster retrieval
- ✅ **Real-time Communication**: SignalR integration for bi-directional communication
- ✅ **API Protection**: Rate Limiting with configurable policies for DDoS prevention
- ✅ **Deployment**: Docker

System Architecture

The system follows the **Clean Architecture** pattern, separating the application into the following layers:

1. **Domain Layer**: Contains the domain entities, value objects, interfaces, and business logic.
2. **Application Layer**: Handles the application logic, CQRS commands, and queries.
3. **Infrastructure Layer**: Implements the interfaces defined in the Application layer, such as database access and external services.
4. **Web Layer**: The API layer that communicates with the front-end, providing RESTful endpoints.
5. **Contracts Layer**

Layers:

- **Core**:
  - Entities (Product)
  - Enums (ProductStatus)
  - Value Objects (Tag, Address)
  - Interfaces (IProductRepository)
- **Application**:
  - Commands (AddProductCommand, UpdateProductCommand,)
  - Queries (GetProductByIdQuery, GetProductsByTagQuery,)
  - DTOs (ProductDto)
  - Validators (ProductValidator)
- **Infrastructure**:
  - Repositories (ProductRepository)
  - MongoDbContext
- **Web (API)**:
  - Controllers (ProductController)
  - API Configuration (Swagger UI, CORS)

<u>File Structure</u>

- Controllers
- Middleware
- Authentication & Authorization (JWT RSA)
- API Documentation (Swagger UI)
- Exception Handling Middleware
- JWT Token Middleware

ShgEcom.Application (Business Logic Layer)

- Commands (CQRS - Create/Update/Delete)
  - Product Commands (AddProductCommand, UpdateProductCommand, DeleteProductCommand)
  - Order Commands (CreateOrderCommand, CancelOrderCommand)
  - Customer Commands (RegisterCustomerCommand, UpdateCustomerCommand)
- Queries (CQRS - Read)
  - GetProductByIdQuery, GetAllProductsQuery
  - GetOrdersByCustomerQuery
- DTOs (Data Transfer Objects)
- Interfaces (Abstractions)
- MediatR for Command Handling
- Validation (FluentValidation)
- Authentication Service (JWT Implementation)
- Token Service (JWT Generation & Validation)

ShgEcom.Contracts

- Contracts for Request/Response Models
- Authentication Contracts (Login/Register)
- JWT Token Response Model

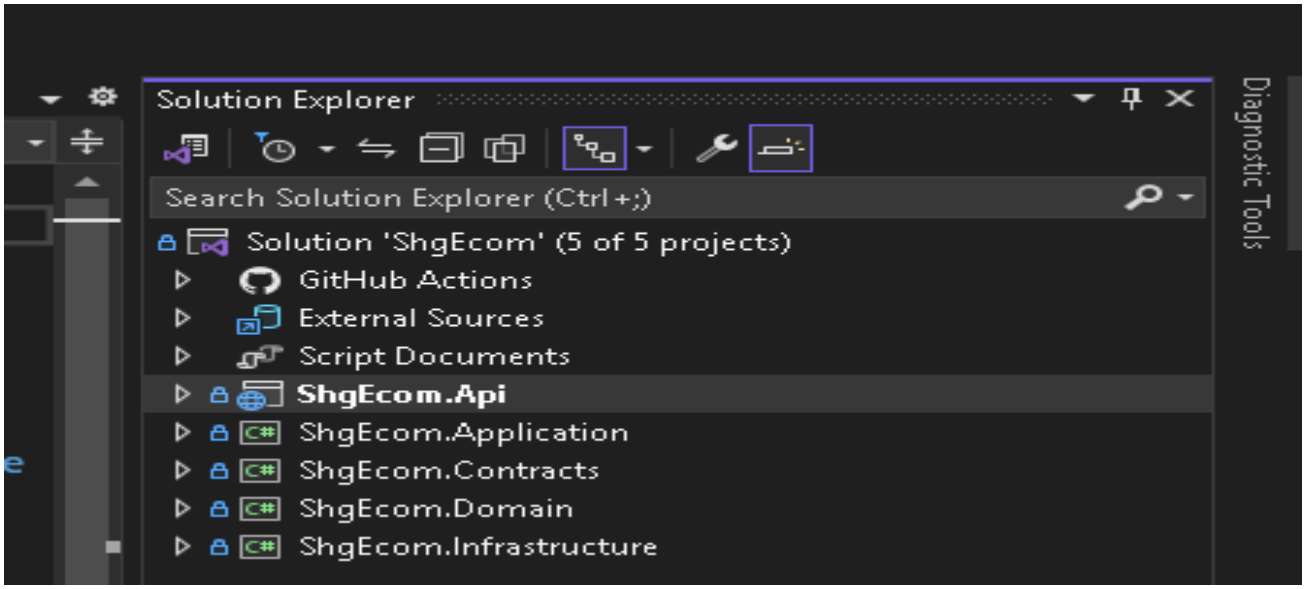
ShgEcom.Domain (Core Entities & Business Rules)

- Entities (Product, User)
- Domain Events

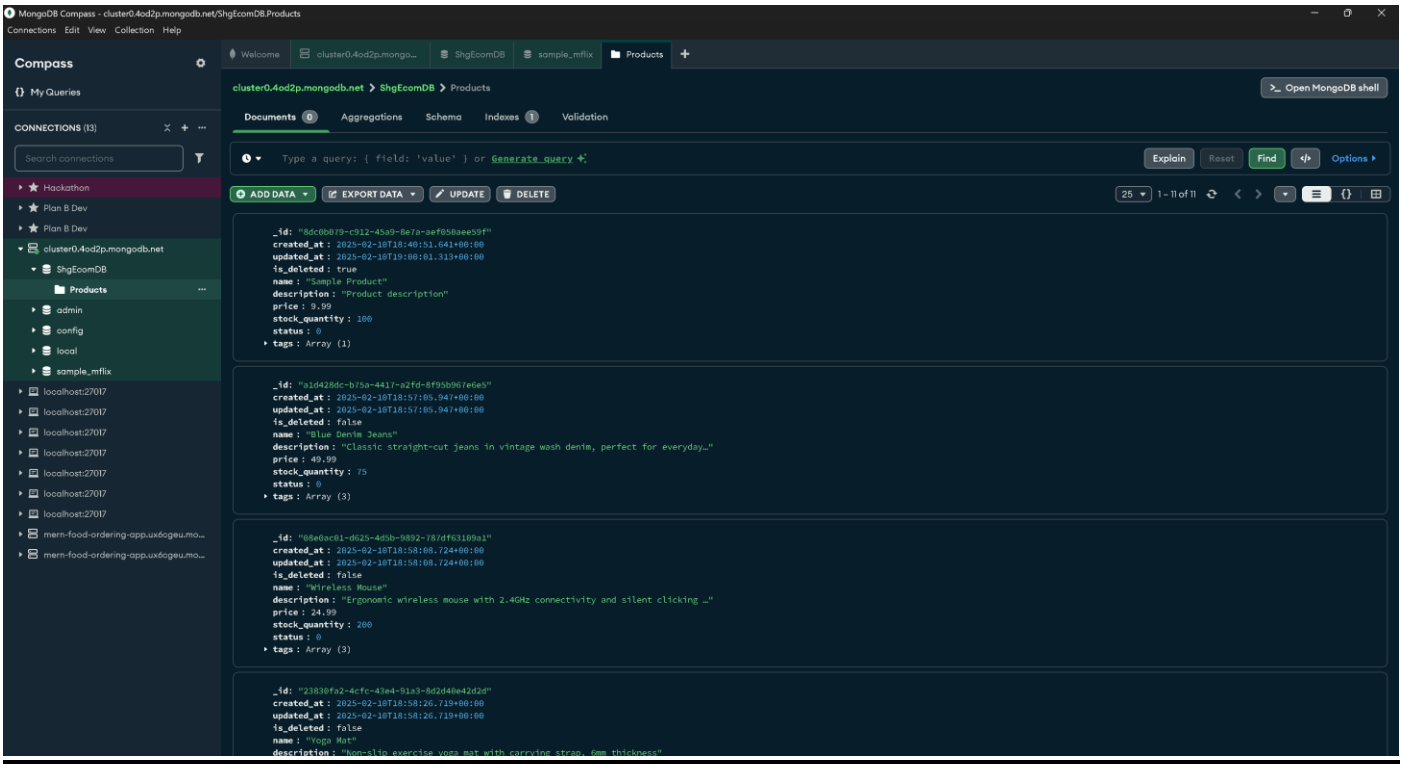
ShgEcom.Infrastructure (Persistence & External Services)

- EF Core Repository Pattern
- MongoDB Implementation
- Caching
- Email Notification Service
- Secure User Management & Hashing

- Token Validation & Security Middleware
- Database Context & Migrations
- Repository Implementations (MongoDB)



Database Design



The application uses **MongoDB** as the primary database to store all entities. The key entities and their respective schemas are:

Base Enity Schema:

```
public abstract record BaseEntity
{
    [BsonId]
    [BsonRepresentation(BsonType.String)]
    public Guid Id { get; set; }

    [BsonElement("created_at")]
    public DateTime CreatedAt { get; set; }

    [BsonElement("updated_at")]
    public DateTime UpdatedAt { get; set; }

    [BsonElement("is_deleted")]
    public bool IsDeleted { get; set; }

    // Constructor for setting default values
    protected BaseEntity()
    {
        Id = Guid.NewGuid();  // Automatically generate a new ObjectId for the entity
        CreatedAt = DateTime.UtcNow;
        IsDeleted = false;
    }

    public void SoftDelete()
    {
        IsDeleted = true;
        UpdatedAt = DateTime.UtcNow;
    }

    public void Restore()
    {
        IsDeleted = false;
        UpdatedAt = DateTime.UtcNow;
    }
}
```

Product Schema:

```
public class Product : BaseEntity
{
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public int StockQuantity { get; set; }
    public ProductStatus Status { get; set; }
    public List<string> Tags { get; set; }
}
```

MongoDbContext:

```
public class MongoDbContext
{
    private readonly IMongoDatabase _database;

    public MongoDbContext(IConfiguration configuration)
```

```
    {
        var client = new MongoClient(configuration["MongoDbSettings:ConnectionString"]);
        _database = client.GetDatabase(configuration["MongoDbSettings:DatabaseName"]);
    }

    public IMongoCollection<Product> Products => _database.GetCollection<Product>("Products");
    public IMongoCollection<Client> Clients => _database.GetCollection<Client>("Clients");
    public IMongoCollection<Instructor> Instructors => _database.GetCollection<Instructor>("Instructors");
}
```

Cache Implementation

Cache Storage

The system uses IMemoryCache from .NET Core to store frequently accessed data.

- Cache duration: **10 minutes**
- Cache key: **products_list**
- Storage location: **Application memory (RAM)**

Cache Constants

Cache-related configurations are stored in a dedicated class:

```
namespace ShgEcom.Application.Common.Constants

{
    public static class CacheConstants
    {
        public static readonly TimeSpan CacheDuration = TimeSpan.FromMinutes(10);
        public const string ProductsListKey = "products_list";
    }
}
```

Cache Usage in Product Service

Retrieving Products with Cache

The GetAllProductsQueryHandler first checks if data exists in cache before querying the database.

```
csharp
CopyEdit
public class GetAllProductsQueryHandler(IProductRepository productRepository, IMemoryCache memoryCache)
    : IRequestHandler<GetAllProductsQuery, ErrorOr<List<ProductResult>>>
{
    public async Task<ErrorOr<List<ProductResult>>> Handle(GetAllProductsQuery request, CancellationToken cancellationToken)
    {
        if (!memoryCache.TryGetValue(CacheConstants.ProductsListKey, out List<Product>? products))
        {
            products = await productRepository.GetAllAsync();
            var cacheOptions = new MemoryCacheEntryOptions
            {
                AbsoluteExpirationRelativeToNow = CacheConstants.CacheDuration
            };
            memoryCache.Set(CacheConstants.ProductsListKey, products, cacheOptions);
        }
```

```
        if (products == null || products.Count == 0)
        {
            return Domain.Errors.Errors.Product.NoProductsFound;
        }

        return products.Select(product => new ProductResult(product)).ToList();
    }
}
```

Cache Invalidation

Clearing Cache on Data Change

To maintain data consistency, cache is cleared whenever new products are added or updated.

```
public class AddProductCommandHandler(IProductRepository productRepository, IMemoryCache memoryCache)

    : IRequestHandler<AddProductCommand, ErrorOr<ProductResult>>
{
    public async Task<ErrorOr<ProductResult>> Handle(AddProductCommand request, CancellationToken cancellationToken)
    {
        var product = new Product
        {
            Name = request.Name,
            Price = request.Price,
            Stock = request.Stock
        };

        await productRepository.AddAsync(product);

        // Clear cache when new product is added
        memoryCache.Remove(CacheConstants.ProductsListKey);

        return new ProductResult(product);
    }
}
```

CQRS Pattern Implementation

The system follows Command Query Responsibility Segregation (CQRS) to separate the read and write logic:

- Commands: These represent requests that modify data. For example:
    - AddProductCommand
    - UpdateProductCommand
    - SoftDeleteProductCommand
- Queries: These represent requests to retrieve data without modifying it. For example:
    - GetProductsByTagQuery
    - GetProductByIdQuery

Each command and query is handled by a respective Handler, which encapsulates the logic for executing the command or query. The Mediator pattern is used to dispatch commands and queries.

Endpoints

The API exposes the following endpoints:
Products:

- **GET /api/products**: Get all products.
- **GET /api/products/{id}**: Get product by ID.
- **POST /api/products**: Create a new product.
- **PUT /api/products/{id}**: Update an existing product.
- **DELETE /api/products/{id}**: Soft delete a product.
- **GET /api/products/tag/{tag}**: Get products by tag.

Authentication & Authorization

JWT Authentication: Secure endpoints using JSON Web Tokens (JWT). Only authenticated users can access sensitive data or make changes to the database.

MongoDB Integration

The application uses MongoDB to store products, clients, instructors, and other data. The following MongoDB operations are implemented:

- **CRUD Operations** for products, clients, and instructors.
- **Soft Deletion**: Data can be marked as "deleted" but not permanently removed, allowing for recovery.
- **Tag Filtering**: Products can be filtered based on tags

Swagger Documentation

Swagger UI is configured to provide an interactive API documentation. The application automatically generates and displays the API documentation, which includes:

- Endpoint descriptions
- Request and response examples
- Status codes
- Authentication details (JWT)

CORS Support

Cross-Origin Resource Sharing (CORS) is enabled to allow communication between the front-end application and the back-end API, even if they are hosted on different domains.

Error Handling

Error handling is implemented using standard HTTP status codes and custom error messages:

- **400 Bad Request**: Invalid input data.
- **401 Unauthorized**: Invalid or missing authentication.
- **404 Not Found**: Resource not found.
- **500 Internal Server Error**: General server errors.
- **429 Error** : RateLimit Error

Real-time Communication with SignalR

The application leverages SignalR to provide seamless real-time communication capabilities, significantly enhancing user experience and data synchronization. The implementation includes dedicated hubs for different functionalities:

1. **ProductHub**: Handles real-time inventory updates, ensuring all connected clients receive immediate notifications when product quantities change. This prevents overselling and maintains data consistency across all user sessions. The hub broadcasts updates to specific user groups based on their roles and subscriptions.
2. **NotificationHub**: Manages system-wide notifications, including order status changes, price updates, and system alerts. It implements a priority-based notification system where critical updates are pushed immediately while less urgent notifications are queued and batched.

The SignalR implementation includes robust error handling and connection management:

```
public class ProductHub : Hub
{
   private readonly ILogger<ProductHub> _logger;

   public override async Task OnConnectedAsync()
   {
      await Groups.AddToGroupAsync(Context.ConnectionId, "ProductWatchers");
      await base.OnConnectedAsync();
   }

   public async Task BroadcastProductUpdate(ProductUpdateDto update)
   {
      await Clients.Group("ProductWatchers").SendAsync("ReceiveProductUpdate", update);
   }
}
```

API Protection with Rate Limiting

The rate limiting implementation provides comprehensive protection against API abuse while ensuring optimal resource utilization. The system implements multiple rate limiting strategies:

1. **Fixed Window Rate Limiting**:
    - Limits requests to a fixed number within a specified time window
    - Resets completely at the end of each window
    - Ideal for APIs with predictable, steady traffic patterns
    -

```
builder.Services.AddRateLimiter(options =>
{
   options.AddFixedWindowLimiter("fixed", options =>
   {
      options.PermitLimit = 100;
      options.Window = TimeSpan.FromMinutes(1);
      options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
      options.QueueLimit = 2;
   });
});
```

2. **Sliding Window Rate Limiting**:

    - Provides smoother rate limiting by gradually sliding the window
    - Prevents sudden traffic spikes at window boundaries
    - Better for APIs requiring more consistent request distribution

```
builder.Services.AddRateLimiter(options =>
{
   options.AddSlidingWindowLimiter("sliding", options =>
   {
      options.PermitLimit = 100;
      options.Window = TimeSpan.FromMinutes(1);
      options.SegmentsPerWindow = 6;
      options.QueueLimit = 2;
   });
});
```

3. **Token Bucket Rate Limiting**:
    - Allows for burst traffic while maintaining long-term rate limits
    - Tokens replenish at a fixed rate
    - Ideal for APIs with occasional traffic spikes

```
builder.Services.AddRateLimiter(options =>
{
   options.AddTokenBucketLimiter("token", options =>
   {
      options.TokenLimit = 100;
      options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
      options.QueueLimit = 2;
      options.ReplenishmentPeriod = TimeSpan.FromSeconds(1);
      options.TokensPerPeriod = 20;
   });
});
```

The rate limiting implementation includes:

- **Custom Response Handling**: When limits are exceeded, clients receive detailed response headers:

Copy
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 60

- **Dynamic Rate Adjustment**: Rates can be adjusted based on:
    - Time of day
    - Server load
    - User subscription level
    - API endpoint importance
- **Monitoring and Metrics**: The system tracks:
    - Request counts per endpoint
    - Rate limit hits
    - Response times
    - Queue lengths

This comprehensive rate limiting strategy ensures our API remains responsive and available while protecting against:

- DDoS attacks
- Brute force attempts
- Scraping abuse
- Resource exhaustion

How To Run the Project

This section provides a **step-by-step guide** to setting up, running, and testing the **Shgardi Backend Task** locally

1. Prerequisites
Before running the project, ensure you have the following installed:
✅ **.NET SDK 8.0+** – Download Here
✅ **MongoDB (Database Server)** – Download Here
✅ **Docker (For containerized deployment)** – Download Here
✅ **Postman or cURL** – For testing API endpoints
✅ **Visual Studio 2022+ / VS Code** – IDE for development

2. Clone the Repository
```
git clone https://github.com/jaeson-karter-joseph/ShgEcom_Backend_Task.git
cd ShgEcom_Backend_Task
```

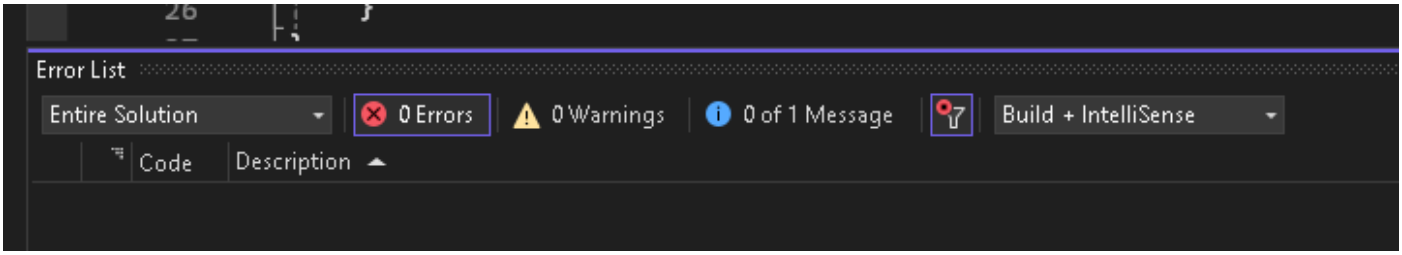
3. Configure AppSettings
```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "MongoDbSettings": {
    "ConnectionString": "ConnectionString",
    "Database": "ShgEcomDB",
    "ProductCollectionName": "Products"
  },
  "JwtSettings": {
    "Secret": "YourSuperSecretKeyForJwt ",
    "ExpiryMinutes": 60,
    "Issuer": "ShgEcom",
    "Audience": "ShgEcom"
  }
}
```

My Temporary Cloud Connection String:
mongodb+srv://jkjoseph2023:RrK3Vj9WdSQWH2Bn@cluster0.4od2p.mongodb.net/

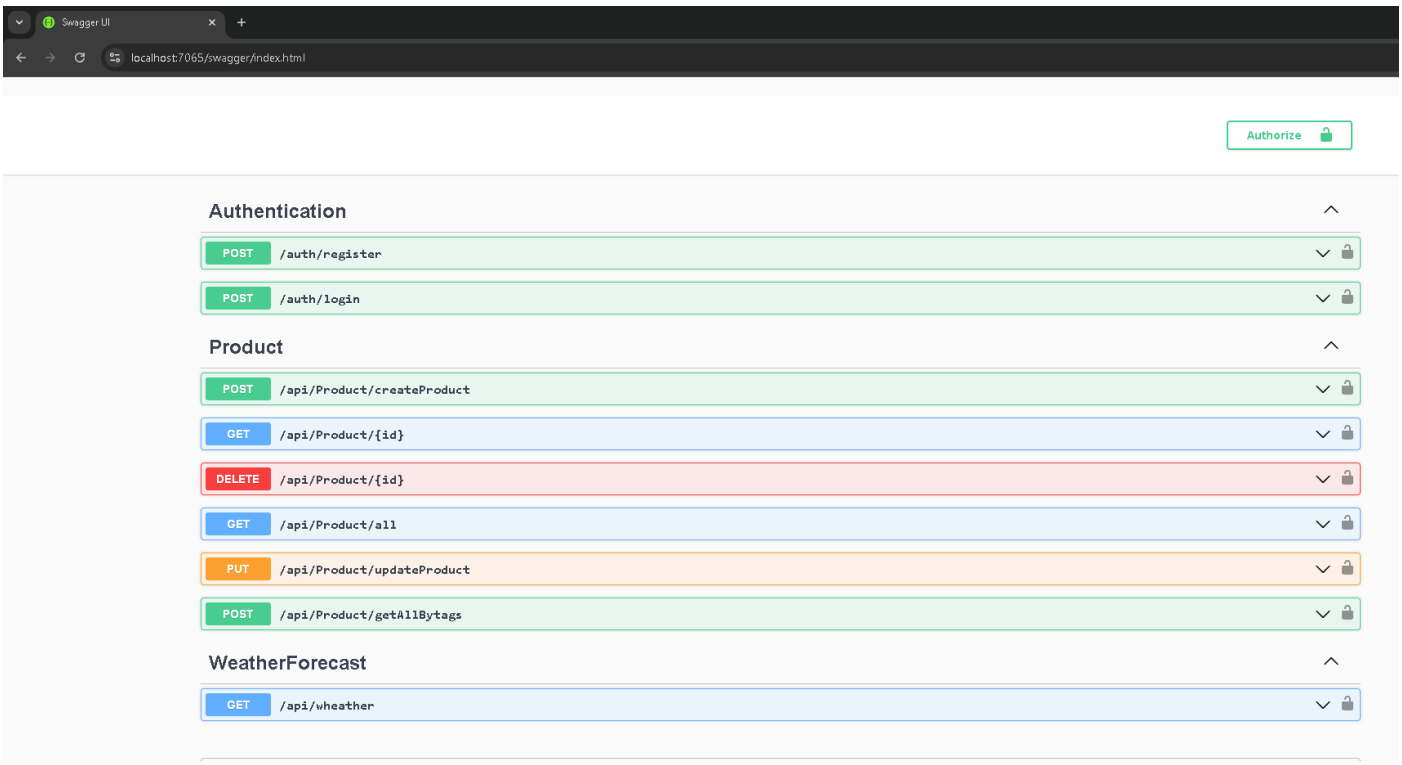
4.Restore Dependencies
```
dotnet restore
```

5. Run the Application
```
dotnet run --project ShgEcom.API
```

Code Quality

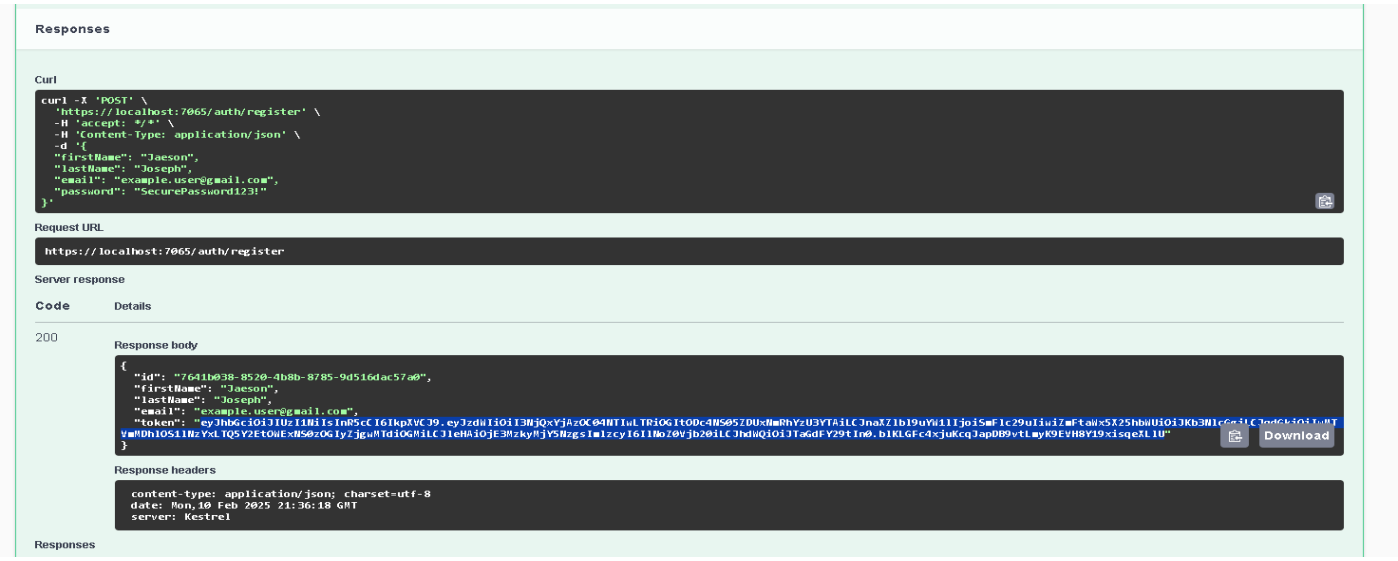


Swagger UI

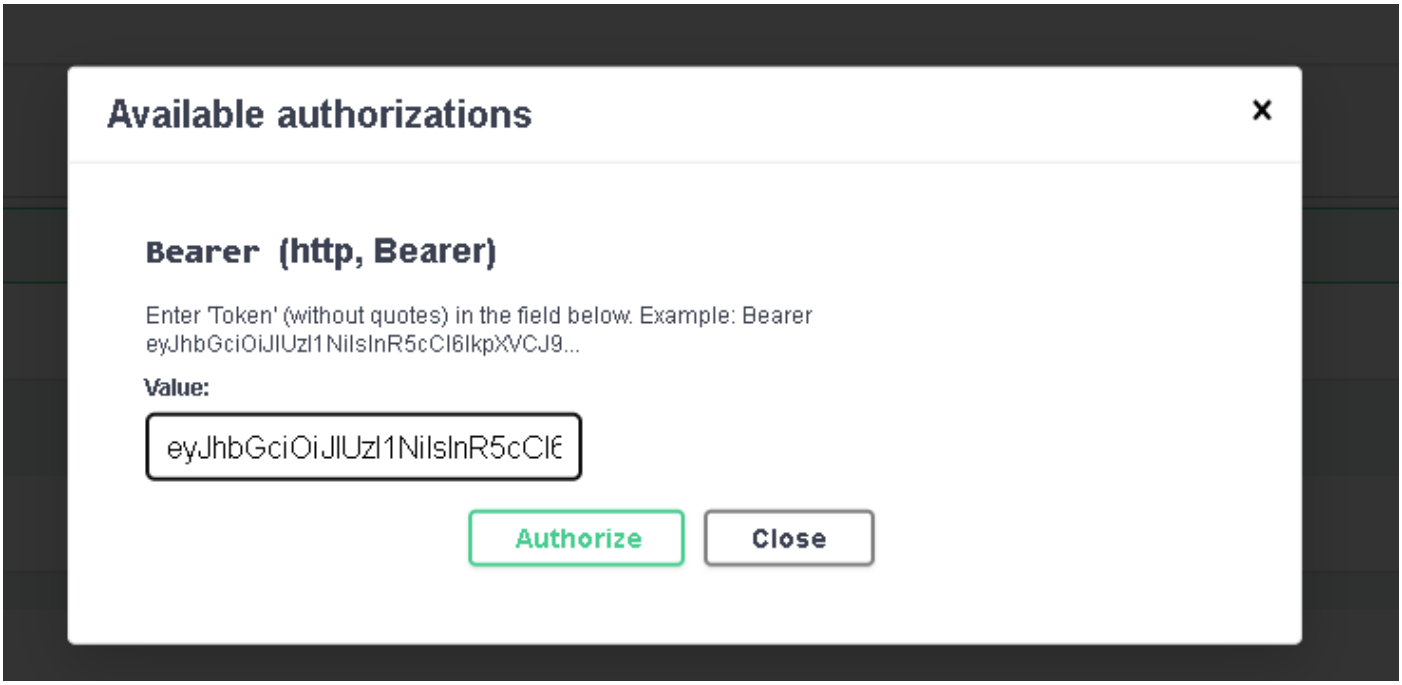


Sample API Calls

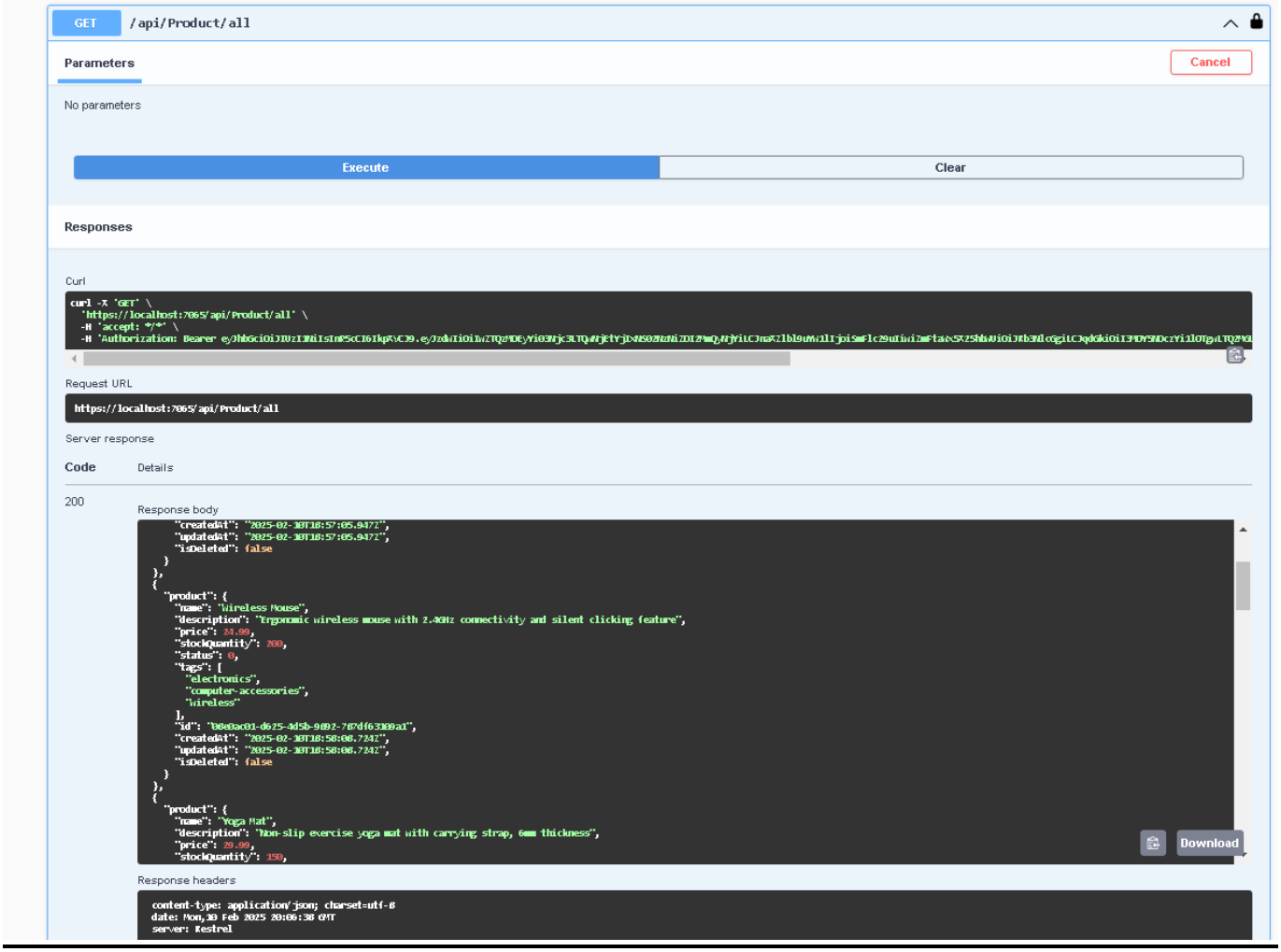
1.Get The Token

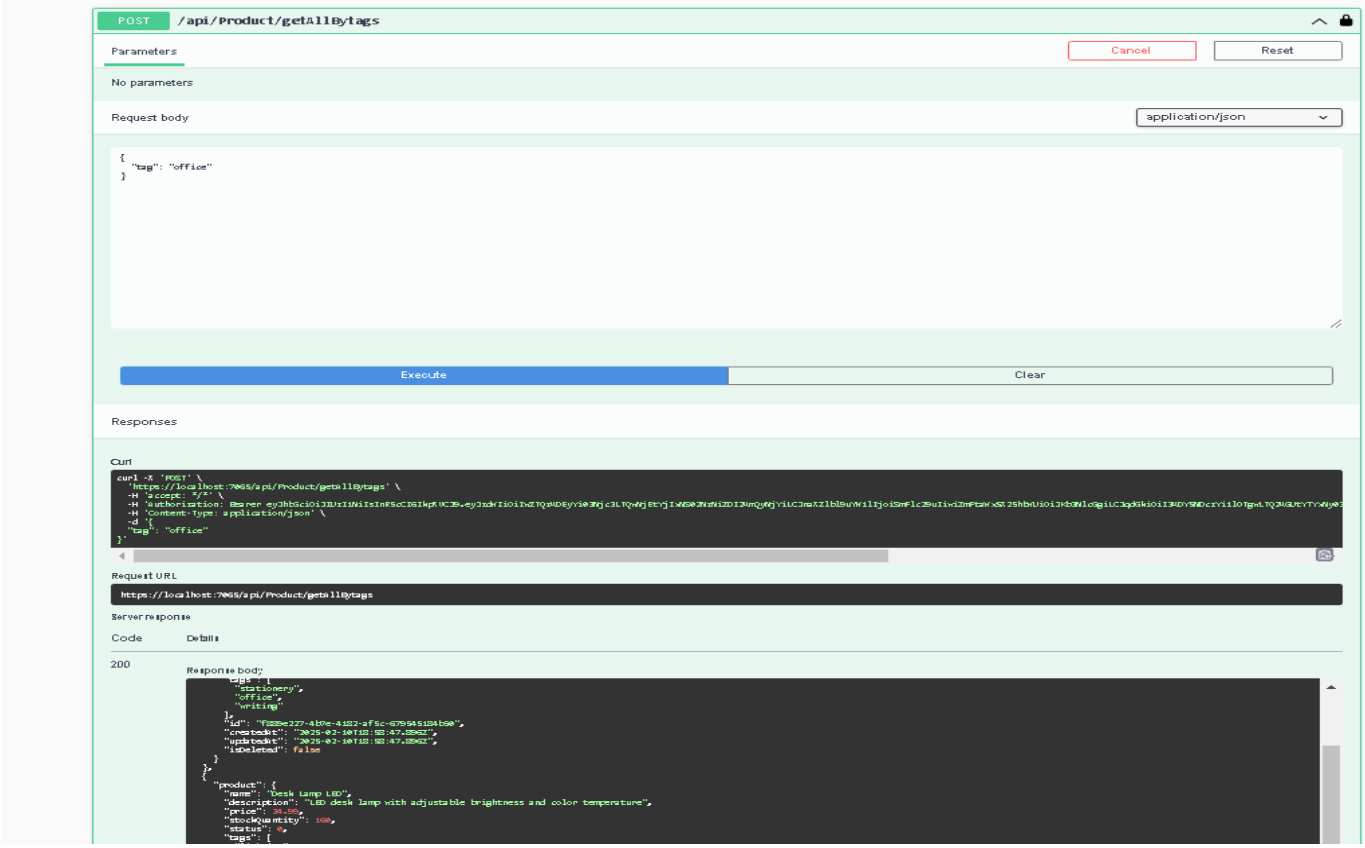


2. Authorize



3. Use APIS




POST
/api/Product/getAllBytags
Parameters
Cancel
Reset
No parameters
Request body
application/json
{
"tag": "office"
}
Execute
Clear
Responses
Curl
curl -x 'POST' \
'https://localhost:7065/api/Product/getAllBytags' \
-H 'accept: */*' \
-H 'Content-Type: application/json' \
-d '{
"tag": "office"
}'
Request URL
https://localhost:7065/api/Product/getAllBytags
Server response
Code
Details
200
Response body
"tags": [
"stationery",
"office",
"writing"
],
"id": "f885e227-4b7e-4182-af5c-67954518b5a",
"isDeleted": false
},
"product": {
"name": "Desk Lamp LED",
"description": "LED desk lamp with adjustable brightness and color temperature",
"stockQuantity": 100,
"status": 0,
"tags": [

The End