Project Documentation

E-commerce Application with Docker

Overview

This documentation provides comprehensive information about the e-commerce platform, including its architecture, features, setup instructions, and technical details.

Table of Contents

Introduction

The e-commerce platform is a containerized web application that allows users to authenticate, manage their profiles, and perform CRUD operations on products with images. This project demonstrates the implementation of Docker containerization for a multi-tier application with a database, backend API, and frontend UI.

Project Goals

- Create a robust e-commerce platform with modern technologies
- Demonstrate the implementation of Clean Architecture principles
- Showcase Docker containerization for .NET and Flutter applications
- Provide a practical example of JWT authentication and Entity Framework Core
- Create a responsive user interface that works across multiple platforms

Technical Stack

Backend

- **Framework**: ASP.NET Core 8.0
- **Language**: C#
- **Database**: Microsoft SQL Server 2019
- **ORM**: Entity Framework Core 9.0
- **Authentication**: JWT (JSON Web Tokens)
- **API Documentation**: Swagger/OpenAPI
- **Containerization**: Docker

Frontend

- **Framework**: Flutter
- **Language**: Dart
- **State Management**: BLoC Pattern
- **HTTP Client**: Dio
- **Image Handling**: Multi-platform support

DevOps

- **Containerization**: Docker
- **Orchestration**: Docker Compose
- **Version Control**: Git
- **Image Repository**: Docker Hub

Architecture

The application follows Clean Architecture principles, separating concerns into distinct layers:

Backend Architecture

1. **Presentation Layer (API Controllers)**
   - Account Controller: Handles user authentication and profile management
   - Item Controller: Manages product operations
2. **Domain Layer**
   - Entities: User, Product
   - Interfaces: Repositories, Services
3. **Data Access Layer**

- Entity Framework Context
- Repository Implementations
- Database Migrations

Frontend Architecture

1. **Presentation Layer**
   - Screens: Login, Register, Product List, Product Details, etc.
   - Widgets: Reusable UI components
2. **Business Logic Layer**
   - BLoCs: Authentication, Product Management
   - Models: Data structures
3. **Data Layer**
   - Repositories: User, Product
   - API Clients: HTTP communication with backend

Setup and Deployment

Prerequisites

- Docker and Docker Compose
- Git
- Internet connection for pulling Docker images

Local Development Setup

1. Clone the repository:

   ```bash
   git clone https://github.com/mohamed12344556/E-Commerce-App-With-Docker.git
   cd ecommerce-project
   ```

2. Create required configuration files:
   - `init-db.sql`: Initial database setup script
   - `entrypoint.sh`: Application startup script
   - `docker-compose.yml`: Service orchestration
   - `Dockerfile`: Backend image definition
3. Launch the application:

bash

```
docker-compose up --build
```

4. Access the application:
   - Backend API: http://localhost:5163/swagger

Production Deployment

1. Modify environment variables in `docker-compose.yml`:

yaml

```
environment:
  - ASPNETCORE_ENVIRONMENT=Production
  - ConnectionStrings__conStr=Server=sql-server;Database=Ecommerce;User Id=sa;Password=Your
```

2. Push Docker images to Docker Hub:

bash

```
docker login
docker tag ecommerce-backend yourusername/ecommerce-backend:latest
docker push yourusername/ecommerce-backend:latest
```

3. On the production server, pull and run the images:

bash

```
docker-compose -f docker-compose.production.yml up -d
```

API Documentation

Authentication Endpoints

Endpoint	Method	Description	Request Body	Response
`/api/Account/Register`	POST	Register new user	`{ "userName": "string", "email": "string", "password": "string", "phoneNumber": "string" }`	`200 OK`
`/api/Account/Login`	POST	User login	`{ "email": "string", "password": "string" }`	`200 OK` + JWT Token
`/api/Account/UpdateUser`	PUT	Update user profile	`{ "userName": "string", "email": "string", "password": "string", "phoneNumber": "string" }`	`200 OK`
`/api/Account/GetUserDataById`	GET	Get user data	None	User Data
`/api/Account/GetAllUsers`	GET	Get all users	None	List of Users

Product Endpoints

Endpoint	Method	Description	Request Body	Response
`/api/Item/GetItems`	GET	Get all products	None	List of Products
`/api/Item/AddItem`	POST	Add new product	Form data with image	`200 OK`
`/api/Item/UpdateItem/{id}`	PUT	Update product	Form data with image	`200 OK`
`/api/Item/DeleteItem/{id}`	DELETE	Delete product	None	`200 OK`

Database Schema

Tables

1. **AspNetUsers**
   - Id (PK)
   - UserName
   - NormalizedUserName
   - Email
   - NormalizedEmail
   - EmailConfirmed
   - PasswordHash
   - SecurityStamp

- ConcurrencyStamp
- PhoneNumber
- PhoneNumberConfirmed
- TwoFactorEnabled
- LockoutEnd
- LockoutEnabled
- AccessFailedCount

2. **AspNetRoles**
   - Id (PK)
   - Name
   - NormalizedName
   - ConcurrencyStamp
3. **Items**
   - Id (PK)
   - Title
   - Description
   - ImagePath
   - CreatedAt
   - UpdatedAt

Security Considerations

- **Authentication**: JWT tokens with proper expiration
- **Password Storage**: Hashed and salted using ASP.NET Core Identity
- **SQL Injection**: Prevented through Entity Framework parameterized queries
- **CORS**: Configured to allow specific origins
- **Docker Security**: Non-root users for containers
- **Secrets Management**: Environment variables for sensitive information

Performance Optimization

- **Database**: Entity Framework Core with optimized queries
- **API Responses**: Properly structured DTOs
- **Docker**: Multi-stage builds to reduce image size

- **Connection Pooling**: Configured for SQL Server connections
- **Error Resilience**: Retry mechanisms for transient failures

Known Issues and Solutions

Issue: SQL Server Connection Problems

Symptoms: Backend cannot connect to SQL Server, error messages about "A network-related or instance-specific error"

Solution:

1. Ensure SQL Server container is running: `docker ps`
2. Check SQL Server logs: `docker logs ecommerce-sql-server`
3. Verify connection string in both docker-compose.yml and appsettings.json
4. Add `EnableRetryOnFailure()` to DbContext configuration
5. Ensure the ACCEPT_EULA=Y environment variable is set

Issue: Database Tables Not Created

Symptoms: "Invalid object name 'AspNetUsers'" errors

Solution:

1. Run Entity Framework migrations: `dotnet ef database update`
2. Or create tables manually using init-db.sql
3. Modify Program.cs to apply migrations automatically at startup

Future Enhancements

1. Implement product categories and search functionality
2. Add payment processing capabilities
3. Enhance security with HTTPS and additional authentication options
4. Implement caching layer for improved performance
5. Add comprehensive testing (unit tests, integration tests)
6. Set up CI/CD pipeline for automated deployment
7. Implement monitoring and logging solutions