EEC 626 Software Engineering Project

Developers Guide

Inventory Management Web Application

Wongpiwat Sa Ngiam - 2883857
Biswadeep Mazumder – 2860920
Uday Shankar Boddupalli - 2868122

1. Introduction

Project Overview
This project is an Inventory Management Web Application and primarily built to streamline inventory management processes for small to medium-sized businesses. The backend, implemented using .NET 8 Web API, serves as the core of the application, managing all server-side logic, data processing, and API endpoints that facilitate seamless interaction between the database and frontend. The backend ensures secure, efficient, and scalable operations for managing orders, products, and suppliers. For the frontend, this project is implemented using React, a library for building dynamic web features and responsive user interfaces. that enables fast, modular, and maintainable component-based development. Next.js, a web framework, is utilized to support server-side rendering (SSR) and static site generation (SSG), enhancing performance, scalability, and SEO. Material-UI, a design component library, is used to craft visually appealing and responsive user interfaces, providing a consistent look and feel across the application while adhering to modern design principles. Together, these technologies ensure the application is both visually engaging and technically robust, making it a reliable solution for modern inventory management.

Technologies Used
1. **Backend**:
   - .NET 8 Web API Framework
   - Entity Framework Core for ORM (Object-Relational Mapping)
2. **Frontend**:
   - React 18 for web features and user interfaces
   - Next.js 14 for web framework
   - Material UI for styling components
3. **Database**
   - Azure SQL Database
4. **API Testing**:
   - Swagger for API documentation and testing
   - Postman for manual endpoint testing
5. **Hosting and Deployment**:
   - Azure Web App Service for hosting the backend
   - Azure DevOps for CI/CD pipelines
6. **Development Tools**:
   - Visual Studio Code IDE for backend development
   - WebStorm for frontend development
   - GitHub for version control and repository management

Project Architecture
Backend: The .NET Web API backend follows a clean architecture pattern:
1. **Controller Layer**:
   - Handles HTTP requests and routes them to appropriate services.
   - Example Controllers: OrderController, ItemController, SupplierController.
2. **Service Layer**:
   - Implements business logic and performs operations like validation, data transformation, and process management.
   - Services communicate with the repository layer for data access.
3. **Repository Layer**:
   - Provides an abstraction over Entity Framework Core for database operations.
   - Ensures a clean separation between data access and business logic.
4. **Models and DTOs**:
   - Models represent the database entities (e.g., Order, Item, Supplier).

   - DTOs (Data Transfer Objects) shape the data returned to and received from the API.
5. **Database**:
   - Azure SQL Database serves as the persistent data storage.
   - The database schema is mapped to C# models using Entity Framework Core in a database-first approach.

Frontend: The Next.js framework follows a modular and scalable architecture pattern. It leverages server-side rendering (SSR) and static site generation (SSG) to optimize performance and SEO. The architecture typically includes:

1. **Pages and Routing**:
   - uses a file-based routing system where each file in the pages directory corresponds to a route in the application.
2. **Components**:
   - Reusable UI components are organized in the components directory.
3. **Static Assets**:
   - Static files like images and fonts are stored in the public directory.
4. **Styling**:
   - Supports styling options including CSS frameworks like Tailwind CSS and Material UI
5. **State Management**:
   - Integrates with state management libraries like React Context API and hooks.
6. **Configuration**:
   - Custom configurations can be added in the next.config.js file.
7. **Build and Deployment**:
   - Provides build process is managed by the Next.js build system, and deployment is done on platforms like Vercel.

This architecture ensures scalability, maintainability, and efficient data handling, making it easier for future developers to extend or optimize the system.

2. Getting Started

Prerequisites

1. **Operating System**: macOS / Windows / Linux
2. **Software Requirements**:
   - **Visual Studio Code**: Latest version with C# extension for .NET development.
   - **WebStorm**: For TypeScript development.
   - **SQL Server Management Studio (or Azure Data Studio)**: To interact with the Azure SQL database.
   - **Postman**: For testing API endpoints.
   - **Git**: To manage repository operations.
3. **Frameworks and Tools**:
   - **.NET SDK**: Version 8.0.
   - **Entity Framework Core**: Latest stable version compatible with .NET 8.
   - **Azure CLI**: For deployment and Azure resource management.
   - **Node.js**: If the frontend integration is required during testing.

Repository Setup

1. **Clone the Repository**:
   - Open your terminal and run:
     ```
     git clone https://github.com/BiswadeepMazumder/InventoryManagement.git
     ```
   - Navigate to the repository directory:
     ```
     cd InventoryManagement
     ```

2. **Branching Strategy**:
   - The repository is organized with the following branches:
     - **main**: Contains the production-ready code.
     - **backend**: Dedicated to backend development, including all .NET Web API code.
     - **frontend**: Contains the frontend code.

Developers should always branch out from the **backend** branch for feature development and create pull requests for merging back. Same goes for the front end.

Environment Setup
Backend

1. **Configuration Files**:
   - Open the appsettings.json file located in the root directory of the backend.
   - Ensure the following sections are correctly configured:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Your_Azure_SQL_Connection_String"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

   - Replace **"Your_Azure_SQL_Connection_String"** with the actual connection string from your Azure SQL Database.

2. **Database Connection**:
   - Ensure your local machine has access to the Azure SQL Database by configuring your firewall in Azure to allow your public IP.
   - Test the database connection: `dotnet ef dbcontext info`
     If it succeeds, your database connection is correctly configured.

3. **Run the Application Locally**:
   - Open the project in VS Code: `code .`
   - Restore dependencies: `dotnet restore`
   - Build and run the application: `dotnet run`
   - By default, the application will run on https://localhost:5236
     You can use tools like Postman or Swagger UI (enabled) to interact with the APIs on https://localhost:5236/swagger/index.html

4. **Setup for Debugging**:
   - Use VS Code's debugging feature to set breakpoints in your controllers or services and step through code execution.

Frontend

1. **Configuration Files**:
   - Open the .env.local file located in the root directory of the frontend.
   - Ensure the following sections are correctly configured:

```
NEXT_PUBLIC_FIREBASE_CLIENT_API_KEY=YOUR_FIREBASE_API_KEY
NEXT_PUBLIC_FIREBASE_CLIENT_AUTH_DOMAIN=YOUR_FIREBASE_AUTH_DOMAI
N
NEXT_PUBLIC_FIREBASE_CLIENT_PROJECT_ID=YOUR_FIREBASE_PROJECT_ID
NEXT_PUBLIC_FIREBASE_CLIENT_STORAGE_BUCKET=YOUR_FIREBASE_STORAGE
NEXT_PUBLIC_FIREBASE_CLIENT_MESSAGING_SENDER_ID=YOUR_FIREBASE_MSG
NEXT_PUBLIC_FIREBASE_CLIENT_APP_ID=YOUR_FIREBASE_CLIENT_APP_ID
NEXT_PUBLIC_API_ENDPOINT=YOUR_API_ENDPOINT
API_ENDPOINT=YOUR_API_ENDPOINT
NEXT_PUBLIC_STORE_NAME=YOUR_STORE_NAME
STORE_NAME=YOUR_STORE_NAME
```

2. **Run the Application Locally**:
   - Install packages: yarn install
   - Run the application: yarn run dev

3. Project Structure
Backend
Folder Organization

1. **Controllers**:
   - These handle incoming HTTP requests and route them to the appropriate service.
   - **Files**:
     - OrderController.cs(OrderController): Handles order-related operations such as creating, viewing, updating, and canceling orders.
     - ItemController.cs(ItemController): Manages operations for fetching, adding, updating, and deleting items in the inventory.
     - SupplierController.cs(SupplierController): Handles supplier-related tasks like retrieving supplier details.

2. **Models**:
   - Represent the database schema in the application.
   - **Files**:
     - Items.cs(Items): Represents items in the inventory with properties such as ID, name, and stock.
     - Category.cs(Category): Defines categories for inventory items.
     - OrderItems.cs(OrderItems): Links orders to their items.
     - Orders.cs(Orders): Represents customer orders.
     - Supplier.cs(Supplier): Represents suppliers and their details.
     - SupplierToCategory.cs(SupplierToCategory): Maps relationships between suppliers and categories.
     - User.cs(User): Represents application users.

3. **Services**:
   - Contain business logic and data processing.
   - **Files**:
     - CancelOrderDTO.cs(CancelOrderDTO): Defines the structure for canceling orders.

        - CreateOrderDTO.cs(CreateOrderDTO): Represents the structure for creating an order.
        - CreateOrderItemDTO.cs(CreateOrderItemDTO): Used within CreateOrderDTO for individual items.
        - ItemDTO.cs(ItemDTO): Represents items returned in API responses.
        - OrderDTO.cs(OrderDTO): Represents orders sent between frontend and backend.
        - OrderItemDTO.cs(OrderItemDTO): Represents individual items within an order.
        - SupplierDTO.cs(SupplierDTO): Represents supplier details.

4. **Data:**
    - Contains InventoryDbContext for managing database connections and operations (InventoryDbContext).

5. **Other Files**:
    - **Program.cs**(Program): The entry point of the application where services, middleware, and routes are configured.

Key Files

1. Program.cs:
    - Configures dependency injection, middleware, and routing.
    - Key configurations include:
        - DbContext registration using a connection string from appsettings.json or environment variables.
        - Enabling Swagger for API documentation and testing.
        - Adding a **CORS** policy to allow frontend-backend communication.

```
builder.Services.AddDbContext<InventoryDbContext>(options =>
options.UseSqlServer(connectionString).EnableSensitiveDataLogging().LogTo(Console.WriteLine));
```

2. InventoryDbContext.cs:
    - Maps database tables to C# models and defines relationships between them (InventoryDbContext).
    - Example:

```
modelBuilder.Entity<OrderItems>(entity =>
{
   entity.HasKey(e => new { e.OrderId, e.ItemId, e.OrderDate });
   entity.HasOne(d => d.Item)
         .WithMany(p => p.OrderItems)
         .HasForeignKey(d => d.ItemId);
});
```

Dependency Injection

1. **Configured in** Program.cs:
    - Services and DbContext are registered using the built-in dependency injection system.
2. **Purpose**:
    - Promotes modularity by decoupling service implementations from their consumers.
    - Facilitates unit testing by allowing mock implementations to be injected.

Frontend

Project Structure

This is the directory structure for the project:

```
/project-root
├── /node_modules         # Contains all the npm/yarn packages installed for the project
├── /public               # Static files such as images and fonts
│   ├── /images           # Image assets used in the project
│   └── /fonts            # Font files used in the project
├── /src                  # Source code of the project
│   ├── /app              # Main application files
│   │   ├── /dashboard    # Dashboard related components and pages
│   ├── /services         # Service files for handling API requests
│   ├── /styles           # Global and component-specific styles
│   ├── /types            # TypeScript type definitions
│   └── /utils            # Utility functions and helpers
├── .eslintrc.json        # ESLint configuration file
├── .gitignore            # Git ignore file
├── .prettierrc           # Prettier configuration file
├── package.json          # Project dependencies, scripts, and metadata
├── README.md             # Project overview and setup instructions
├── tsconfig.json         # TypeScript configuration file
└── yarn.lock             # Yarn lock file for dependency management
```

Dependencies:

These are the dependencies listed in the package.json file along with their descriptions:

@emotion/react: A library for writing CSS styles with JavaScript.
@emotion/styled: A styled component library built for Emotion.
@hookform/resolvers: Resolvers for react-hook-form validation.
@mui/icons-material: Material Design icons for MUI.
@mui/material: Material-UI components for React.
@mui/system: Low-level utility functions for building custom design systems with MUI.
@mui/x-date-pickers: Date and time pickers for MUI.
@phosphor-icons/react: Phosphor icons for React.
apexcharts: A modern charting library that helps developers to create beautiful and interactive visualizations.
axios: A promise-based HTTP client for the browser and Node.js.
dayjs: A lightweight JavaScript date library for parsing, validating, manipulating, and formatting dates.
firebase: Firebase JavaScript SDK for client-side development.
nanoid: A tiny, secure URL-friendly unique string ID generator.
next: The React framework for production.
react: A JavaScript library for building user interfaces.
react-apexcharts: A React wrapper for ApexCharts.
react-csv: A library to handle CSV file operations in React.
react-dom: This package serves as the entry point to the DOM and server renderers for React.
react-hook-form: A performant, flexible, and extensible form library for React.
react-toastify: A library to add notifications to your React app.
xlsx: A library to parse and write spreadsheet files.
zod: A library to handle schema validation.

4. APIs

Endpoints

The backend API provides a set of endpoints grouped under different controllers to manage orders, items, and suppliers. Below is a list of all API endpoints, their HTTP methods, and their purposes:

OrderController

1. **GET** /api/Order
   - **Purpose**: Retrieve a list of all orders.
2. **GET** /api/Order/ViewOrderDetail/{id}
   - **Purpose**: Retrieve detailed information about a specific order by ID.
3. **POST** /api/Order/CreateOrder
   - **Purpose**: Create a new order with associated order items.
4. **PUT** /api/Order/CancelOrder/{id}
   - **Purpose**: Cancel an existing order by updating its status and adding a cancel comment.
5. **GET** /api/Order/UpcomingOrders
   - **Purpose**: Retrieve a list of upcoming orders (status indicating future fulfillment).
6. **GET** /api/Order/CurrentOrders
   - **Purpose**: Retrieve a list of current orders (status indicating they are in process).
7. **GET** /api/Order/PastOrders
   - **Purpose**: Retrieve a list of past orders (status indicating completion).
8. **GET** /api/Order/GenerateInvoice/{orderId}
   - **Purpose**: Generate an invoice in HTML format for a specific order.

ItemController

1. **GET** /api/Item
   - **Purpose**: Retrieve a list of all items in the inventory.
2. **GET** /api/Item/lowstock
   - **Purpose**: Retrieve a list of items that are low in stock.
3. **GET** /api/Item/{id}
   - **Purpose**: Retrieve details of a specific item by ID.
4. **POST** /api/Item
   - **Purpose**: Add a new item to the inventory.
5. **PUT** /api/Item/{id}
   - **Purpose**: Update details of an existing item.
6. **DELETE** /api/Item/{id}
   - **Purpose**: Delete an item from the inventory.

SupplierController

1. **GET** /api/Supplier/suppliers
   - **Purpose**: Retrieve a list of all suppliers.

Authorization and Authentication
Currently, the API allows anonymous access to all endpoints, as indicated by the [AllowAnonymous] attribute used in the controllers. This means no authentication or authorization is required to access the API.

- **Middleware and Libraries Used**:
    - The project is set up to potentially use authentication and authorization in the future.
    - ASP.NET Core provides built-in middleware for authentication and authorization, which can be configured in Program.cs if needed.

Error Handling

- **Exception Handling Middleware**:
    - Configured in Program.cs to handle unhandled exceptions globally.

```
app.UseExceptionHandler("/error");
```

    - This middleware directs exceptions to a centralized error handling endpoint.
- **Error Responses in Controllers**:
    - Controllers return appropriate HTTP status codes and messages based on the outcome of operations.
    - **Common Status Codes**:
        - **200 OK:** Successful operation.
        - **201 Created:** Resource successfully created.
        - **204 No Content:** Successful operation with no content to return (e.g., after deletion).
        - **400 Bad Request**: Invalid input data or request format.
        - **404 Not Found:** Resource not found.
        - **500 Internal Server Error:** General server error.

Error Handling in Database Operations:

- **Try-Catch Blocks**:
    - Used around database operations to catch exceptions and return appropriate error responses.
- **Concurrency Handling**:
    - In update operations, concurrency exceptions are caught to handle cases where data has changed between fetch and update.
    - **Example**:

```
try {
   await _context.SaveChangesAsync();
}catch (DbUpdateConcurrencyException) {
   if (!ItemExists(id)) {
      return NotFound();
   } else {
      throw;
   }
}
```

Where to Find Relevant Code:

- **Exception Handling Middleware**: Configured in **Program.cs**.
- **Model Validation and Error Responses**: Implemented in controller actions across all controllers.
- **Error Handling in Controllers**: Each controller contains error handling logic specific to its operations.

API Usage Examples
OrderController

1. **GET /api/Order**
   - **Purpose**: Retrieve a list of all orders.
   - **Response** Body

```
[
  {
    "orderId": "OD_1234",
    "orderDate": "2024-11-01T00:00:00",
    "orderName": "Order A",
    "userId": "US01",
    "orderAmount": 200.50,
    "orderStatus": 1,
    "cancelComment": null,
    "orderItems": [
      {
        "itemId": "ITM001",
        "itemName": "Item A",
        "itemCount": 5,
        "totalPrice": 50.00
      }
    ]
  }
]
```

   - **Usage Example**: GET https://your-api.com/api/Order

2. **GET /api/Order/ViewOrderDetail/{id}**
   - **Purpose**: Retrieve detailed information about a specific order by ID.
   - **Request**: None
   - **Response**:

```
{
  "orderId": "OD_1234",
  "orderDate": "2024-11-01T00:00:00",
  "orderName": "Order A",
  "userId": "US01",
  "orderAmount": 200.50,
  "orderStatus": 1,
  "cancelComment": null,
  "orderItems": [
    {
      "itemId": "ITM001",
      "itemName": "Item A",
      "itemCount": 5,
      "totalPrice": 50.00
    }
  ]
}
```

   - **Usage Example**: GET https://your-api.com/api/Order/ViewOrderDetail/OD_1234

---

3. **POST /api/Order/CreateOrder**
   - **Purpose**: Create a new order with associated order items.
   - **Request**:

```
{
  "orderName": "New Order",
  "orderAmount": 150.00,
  "orderItems": [
    {
      "itemId": "ITM001",
      "itemCount": 5,
      "itemName": "Item A",
      "totalPrice": 50.00
    },
    {
      "itemId": "ITM002",
      "itemCount": 10,
      "itemName": "Item B",
      "totalPrice": 100.00
    }
  ]
}
```

- **Response**: {
```
  "orderId": "OD_5678",
  "orderName": "New Order",
  "orderAmount": 150.00
}
```

4. **PUT /api/Order/CancelOrder/{id}**
   - **Purpose**: Cancel an existing order by updating its status and adding a cancel comment.
   - **Request**: {
```
  "orderId": "OD_1234",
  "cancelComment": "Customer requested cancellation."
}
```
   - **Response**: {
```
  "orderId": "OD_1234",
  "orderStatus": 0,
  "cancelComment": "Customer requested cancellation."
}
```

5. **GET /api/Order/UpcomingOrders**
   - **Purpose**: Retrieve a list of upcoming orders.
   - **Request**: None
   - **Response**: [{
```
  "orderId": "OD_5678",
  "orderName": "Order B",
  "orderAmount": 300.00
 }
]
```
6. **GET /api/Order/CurrentOrders**
   - **Purpose**: Retrieve a list of current orders.
   - **Request**: None
   - **Response**:
```
[{
   "orderId": "OD_9876",
"orderName": "Order C",
"orderAmount": 450.00
}
]
```

7. **GET /api/Order/PastOrders**
   - **Purpose**: Retrieve a list of past orders.
   - **Request**: None
   - **Response**:[{

```
"orderId": "OD_3456",
"orderName": "Order D",
"orderAmount": 500.00
}
]
```

---

8. **GET /api/Order/GenerateInvoice/{orderId}**
   - **Purpose**: Generate an invoice in HTML format for a specific order.
   - **Request**: None
   - **Response**: HTML Content
   - **Usage Example**:
     - **cURL**:

```
curl -X GET https://your-api.com/api/Order/GenerateInvoice/OD_1234
```

ItemController

1. **GET /api/Item**
   - **Purpose**: Retrieve a list of all items in the inventory.
   - **Request**: None
   - **Response**:[{

```
  "itemId": "ITM001",
  "itemName": "Item A",
  "itemUnitPrice": 50.00,
  "currentStock": 500,
  "status": 1,
  "categoryCode": "CAT01"
 }
]
```

   - **Usage Example**:
     - **cURL**: curl -X GET https://your-api.com/api/Item

---

2. **GET /api/Item/lowstock**
   - **Purpose**: Retrieve a list of items that are low in stock.
   - **Request**: None
   - **Response**:

```
[
  {
   "itemId": "ITM002",
   "itemName": "Item B",
   "itemUnitPrice": 25.00,
   "currentStock": 100,
   "status": 1,
   "categoryCode": "CAT02"
```

```
    }
  ]
```

- **Usage Example**:
  - **cURL**: curl -X GET https://your-api.com/api/Item/lowstock

---

3. **GET /api/Item/{id}**
   - **Purpose**: Retrieve details of a specific item by ID.
   - **Request**: None
   - **Response**:

```
{
  "itemId": "ITM001",
  "itemName": "Item A",
  "itemUnitPrice": 50.00,
  "currentStock": 500,
  "status": 1,
  "categoryCode": "CAT01"
}
```

4. **POST /api/Item**
   - **Purpose**: Add a new item to the inventory.
   - **Request**:

```
{
  "itemId": "ITM003",
  "itemName": "Item C",
  "itemUnitPrice": 30.00,
  "currentStock": 200,
  "status": 1,
  "categoryCode": "CAT03"
}
```

   - **Response**:

```
{
  "itemId": "ITM003",
  "itemName": "Item C",
  "itemUnitPrice": 30.00,
  "currentStock": 200,
  "status": 1,
  "categoryCode": "CAT03"
}
```

   - **Usage Example**:
     - **cURL**: curl -X POST https://your-api.com/api/Item \

```
-H "Content-Type: application/json" \
-d '{"itemId":"ITM003","itemName":"Item C","itemUnitPrice":30.00,"currentStock":200,"status":1,"categoryCode":"CAT03"}'
```

---

5. **PUT /api/Item/{id}**
   - **Purpose**: Update details of an existing item.
   - **Request**:

```
{
  "itemId": "ITM001",
  "itemName": "Updated Item A",
  "itemUnitPrice": 55.00,
  "currentStock": 450,
  "status": 1,
  "categoryCode": "CAT01"
}
```

- **Response**: 204 No Content
- **Usage Example**:
  - **cURL**: curl -X PUT https://your-api.com/api/Item/ITM001 \

```
-H "Content-Type: application/json" \
-d '{"itemId":"ITM001","itemName":"Updated Item A","itemUnitPrice":55.00,"currentStock":450,"status":1,"categoryCode":"CAT01"}'
```

---

6. **DELETE /api/Item/{id}**
   - **Purpose**: Delete an item from the inventory.
   - **Request**: None
   - **Response**: 204 No Content
   - **Usage Example**:
     - **cURL**: curl -X DELETE https://your-api.com/api/Item/ITM001

SupplierController

1. **GET /api/Supplier/suppliers**
   - **Purpose**: Retrieve a list of all suppliers.
   - **Response**:

```
[
  {
    "supplierName": "Supplier A",
    "supplierAddress": "123 Main St",
    "supplierCity": "Cityville",
    "supplierZipCode": 12345,
    "supplierPhoneNumber": 1234567890
  }
]
```

Swagger Documentation

- **Swagger UI**:
  - The API includes Swagger for interactive documentation and testing.
  - **Accessing Swagger UI**:
    - Navigate to the root URL:

https://eec626softwareproject-ebcgazeehphxgedh.canadacentral-01.azurewebsites.net/swagger/index.html

  - **Features**:
    - View all available endpoints with descriptions.
    - Test endpoints directly from the browser.
    - View request and response models.

Testing the API

- **Using Postman**:
  - Import the API endpoints into Postman for testing.
  - Set up collections for organized testing of different controllers.
  - Test various scenarios, including successful requests and error conditions.

Additional Notes

- **Endpoint Naming Conventions**:
  - Follows RESTful principles with clear and consistent naming.
  - HTTP methods are used appropriately for CRUD operations:
    - GET for retrieval.
    - POST for creation.
    - PUT for updates.
    - DELETE for deletions.
- **Data Transfer Objects (DTOs)**:
  - DTOs are used to define the structure of data sent to and from the API.
  - Located in the **Services** folder.
  - Ensure that only necessary data is exposed, enhancing security and efficiency.

Database
Schema Overview
The database schema for this project is designed to manage key entities such as orders, items, categories, suppliers, and users. Below is a textual representation of the schema with relationships:

1. **Tables**:
    - **Category**:
        - CategoryCode (Primary Key)
        - CategoryName
        - Status
    - **Items**:
        - ItemId (Primary Key)
        - ItemName
        - ItemUnitPrice
        - CurrentStock
        - Status
        - CategoryCode (Foreign Key → Category.CategoryCode)
    - **Orders**:
        - OrderId (Primary Key)
        - OrderDate
        - OrderName
        - UserId (Foreign Key → User.UserId)
        - OrderAmount
        - OrderStatus
        - CancelComment
    - **OrderItems**:
        - Composite Primary Key: (OrderId, ItemId, OrderDate)
        - OrderId (Foreign Key → Orders.OrderId)
        - ItemId (Foreign Key → Items.ItemId)
        - ItemCount
        - ItemName
        - TotalPrice
        - OrderStatus
    - **Supplier**:
        - SupplierId (Primary Key)
        - SupplierName
        - SupplierAddress
        - SupplierCity
        - SupplierZipCode
        - SupplierPhoneNumber
        - SupplierLastOrderDate
    - **SupplierToCategory**:
        - Composite Primary Key: (CategoryCode, SupplierId)
        - CategoryCode (Foreign Key → Category.CategoryCode)
        - SupplierId (Foreign Key → Supplier.SupplierId)
    - **User**:
        - UserId (Primary Key)
        - UserName
        - UserDescription
        - UserRole
        - Status

Database Context

The **InventoryDbContext** in this project serves as the bridge between the .NET application and the database. It leverages Entity Framework Core to enable CRUD operations.

- **File**: InventoryDbContext.cs(InventoryDbContext).
- **Key Responsibilities**:
  - Map database tables to C# models using **DbSet** properties.
  - Configure relationships, primary keys, and constraints using Fluent API in the OnModelCreating method.
  - Provide a session for querying and saving data.
- **Example Configuration**:

```
public virtual DbSet<Orders> Orders { get; set; }
public virtual DbSet<Items> Items { get; set; }
public virtual DbSet<Category> Category { get; set; }
```

- **Relationship Configuration Example**:

```
modelBuilder.Entity<OrderItems>(entity =>
{
   entity.HasKey(e => new { e.OrderId, e.ItemId, e.OrderDate });
   entity.HasOne(d => d.Item).WithMany(p => p.OrderItems)
        .HasForeignKey(d => d.ItemId);
});
```

Migration Process

Entity Framework Core enables developers to manage database schemas via migrations.

1. **Applying Migrations**:
   - Ensure you are in the project directory containing **DbContext**.
   - Run the following command to apply migrations to the database:

     ```
     dotnet ef database update
     ```

   - This applies all pending migrations to the database.
2. **Creating a New Migration**:
   - When schema changes are made, create a new migration to reflect these changes in the database.
   - Steps:
     1. Open the terminal in the project directory.
     2. Run:

        ```
        dotnet ef migrations add <MigrationName>
        ```
3. **Verifying Migrations**:
   - Ensure the Migrations folder is updated with a new migration file.
   - Check the Up and Down methods in the migration file for correctness.
4. **Rolling Back Migrations**:
   - To roll back a migration:

     ```
     dotnet ef database update <PreviousMigrationName>
     ```

6. Development Practices

Coding Standards

To ensure consistency and maintainability, the following coding guidelines were followed:

1. **Naming Conventions**:
   - **Classes**: PascalCase (e.g., OrderController, ItemService).
   - **Methods**: PascalCase (e.g., CreateOrder, GetItemById).
   - **Variables**: camelCase (e.g., itemId, orderAmount).
   - **Constants**: UPPER_SNAKE_CASE (e.g., MAX_ITEM_COUNT).
   - **Database Tables and Columns**: PascalCase (e.g., OrderId, ItemName).
2. **Comments**:
   - Use XML comments (///) for documenting methods and classes.
   - Inline comments are used sparingly to explain non-obvious code logic.
3. **Error Handling**:
   - Wrap database operations in try-catch blocks to handle exceptions gracefully.
   - Log errors using a centralized logging mechanism (e.g., Console.WriteLine for debugging).
4. **Code Structure**:
   - Separate layers: Controllers → Services → Repositories.
   - Use DTOs (Data Transfer Objects) to transfer data between layers.

Instructions to Run Tests:

- Ensure all dependencies are restored:

```
dotnet restore
```

- Run all tests in the solution:

```
dotnet test
```

- View detailed results in the terminal.
- **Optional**: Use Visual Studio Test Explorer to run and debug tests interactively.

Version Control

1. **Commit Message Standards**:
   - Use descriptive commit messages to explain the purpose of changes.

Merge Request Guidelines:

- **Branching Strategy**:
  - Use feature branches for development (e.g., feature/add-order-endpoint).
  - Merge feature branches into backend after peer review.
  - Use main for production-ready code.

7. Deployment

Hosting Environment

1. **Platform**:
   - The application is hosted on **Azure App Service**.
   - The Azure App Service provides a scalable environment to host the .NET Web API backend.
2. **Domain**:
   - The backend is accessible via the domain: https://eec626softwareproject-ebcgazeehphxgedh.canadacentral-01.azurewebsites.net.
3. **Configuration**:
   - **App Settings**: Managed via Azure portal (e.g., database connection strings).
   - **Scaling**: Set to scale automatically based on resource utilization.

Continuous Deployment (CD):

- Deploys the application to Azure App Service after a successful build and test cycle directly from the git hub branch.(backend)

UAT server for testing:

This is hosted on a free hosting platform called MonsterASP.com. this allows us to host the application and perform the UAT testing . This is a manual deployment using the FTP process. The database is also hosted on the same premise.

Deployment Steps

Backend

If manual deployment is required, follow these steps:

1. **Prepare the Application**:
   - Ensure the application builds successfully locally: dotnet build
2. **Create a Publish Package**:
   - Publish the application using the .NET CLI: dotnet publish -c Release -o ./publish
   - This generates a deployable package in the publish folder.
3. **Upload to MonsterASP web Service**:
   - Navigate to the portal.
   - Go to the App Service for the backend.
   - In the **Deployment Center**, select **Upload Deployment Package**.
   - Upload the publish folder contents through FTP.

Restart the App Service:

- After the deployment is complete, restart the App Service to apply changes.

Verify the Deployment:

- Use the Swagger UI (/swagger/index.html) or tools like Postman to test the API endpoints.
- Verify that all endpoints are functioning as expected.

Frontend

1. **Prepare Your Project**
   - Push your project to GitHub, GitLab, or Bitbucket.
   - Ensure it works locally (npm run dev and npm run build).
2. **Sign Up and Log In**
   - Go to vercel.com.
   - Sign up or log in with your GitHub, GitLab, or Bitbucket account.
3. **Connect Your Repository**
   - Click **"New Project"** in the Vercel dashboard.

- Select your repository from GitHub/GitLab/Bitbucket.

4. **Configure Settings**
   - Vercel auto-detects your framework (e.g., Next.js).
   - Set the **Build Command** (npm run build) and **Output Directory** (e.g., .next).
   - Add any **Environment Variables** needed.
5. **Deploy**
   - Click **"Deploy"**.
   - Vercel builds and deploys your app, providing a live URL like https://your-project-name.vercel.app.
6. **Test Your App**
   - Open the live URL to ensure the app works as expected.