CHAPTER 1. WORKING PROCESS

1.1 Scrum Model

The Scrum model is a project management methodology that falls under the Agile framework, designed to help software development teams create high-quality products in short timeframes.

Scrum focuses on enhancing collaboration, flexibility, and rapid feedback from customers.

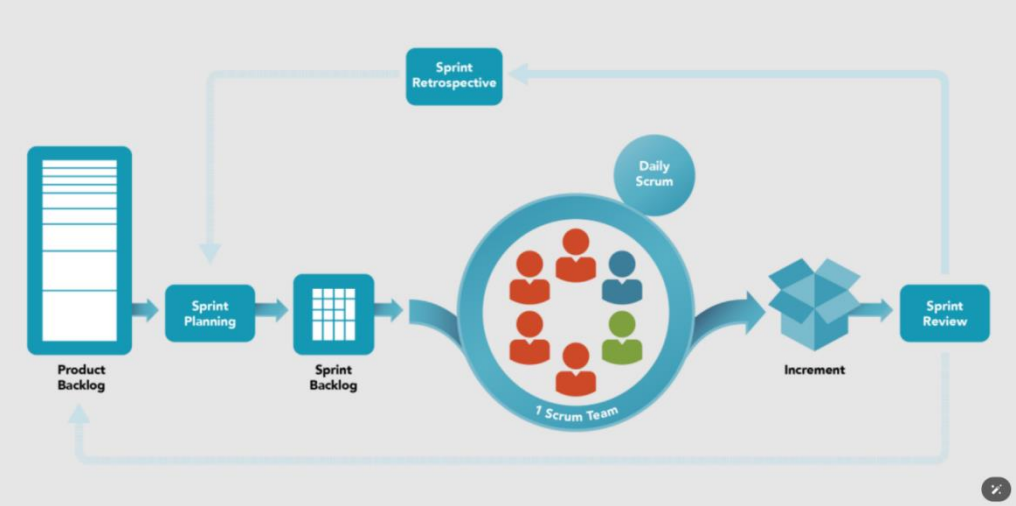


Figure 1. Scrum image

1.2 Scum Team.

The Scrum Team includes the Scrum Master, Product Owner, and Development Team:

- **Scrum Master**: This person supports the team and ensures they follow Scrum principles.
- **Product Owner**: Responsible for managing the product backlog and ensuring the team is working on the highest business priorities.
- **Development Team**: A group of developers, testers, and other technical experts involved in product development.

1.3 Scrum Process

Sprint: A Sprint is a short development cycle, typically lasting from 1 to 4 weeks. During each Sprint, the Scrum team completes a portion of the product or creates a complete product increment.

Sprint Planning: At the start of each Sprint, the Scrum team holds a Sprint Planning meeting to define the tasks to be completed in that Sprint. The Product Owner presents the goals and priorities from the product backlog, and the development team selects the tasks they can complete within the Sprint.

Daily Stand-up: Every day, the Scrum team holds a short meeting called the Daily Meeting. In this meeting, each member reports on their progress, any obstacles they are facing, and their work plan for the day. The goal of this meeting is to enhance transparency and quickly address any arising issues.

Sprint Review: At the end of each Sprint, the Scrum team holds a Sprint Review meeting to present what has been completed during the Sprint. Stakeholders evaluate and provide feedback on the product, helping the team improve and adjust requirements for upcoming Sprints.

Sprint Retrospective: After the Sprint Review, the Scrum team conducts a Sprint Retrospective meeting to assess the team's working process. The team discusses what went well, what needs improvement, and identifies specific actions to enhance efficiency in the next Sprint.

CHAPTER 2. Technology And Job Description

2.1 Detailed Overview of ASP .NET API Development

ASP.NET Core for API Development

ASP.NET Core, a cross-platform framework, was instrumental in developing scalable and high-performance web APIs. Key aspects of ASP.NET Core usage

include:

- **Performance Optimization**: ASP.NET Core is optimized for performance, making it suitable for applications that require handling a large volume of transactions and data.
- **Middleware Integration**: Middleware components in ASP.NET Core facilitated handling HTTP requests and responses efficiently.
- **Dependency Injection**: Built-in support for Dependency Injection simplified the management of services and dependencies, enhancing modularity and testability of the codebase.

2.2 Git and SourceTree for Source Code Management

By using Git and SourceTree, I significantly improved my source code management skills, contributing to the overall success and safe operation of the project during my internship.

- Concepts:
    - Git: A distributed version control system that tracks changes in source code during software development.
    - SourceTree: A graphical user interface (GUI) client for Git, simplifying Git operations.
- Features:
    - Git:
        - Version Control: Tracks and manages changes to the codebase, allowing rollback to previous states.
        - Branching: Supports multiple branches for parallel development, enabling safe experimentation and feature development.
        - Merging: Combines code from different branches, facilitating collaborative development.

        - Commit History: Maintains a detailed log of changes, including who made them and why.
    - SourceTree:
        - User-Friendly Interface: Provides an intuitive GUI for performing Git operations, reducing the need for command-line usage.
        - Visual Representation: Displays branches, commits, and merge conflicts graphically, aiding in understanding the project's history and structure.
        - One-Click Operations: Simplifies common Git tasks like commit, pull, push, and merge with one-click buttons.
        - Conflict Resolution: Assists in resolving merge conflicts through an easy-to-use visual interface.

2.3 Mobile Development with React Native

Technologies and Tools Used

- **React Native**
    - React Native is an open-source framework developed by Facebook that allows developers to build mobile applications for both Android and iOS using JavaScript and React.
    - Throughout my work, I leveraged React Native to create high-performance and visually appealing mobile applications.
- **JavaScript and React**: Enabled the creation of simple and reusable components, utilizing a component-based architecture for efficient UI management.

```jsx
import React from 'react';
import { View, Text, StyleSheet } from 'react-native';

const HelloWorld = () => {
  return (
    <View style={styles.container}>
      <Text style={styles.text}>Hello, React Native!</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  text: {
    fontSize: 20,
    color: '#333',
  },
});

export default HelloWorld;

```

Figure 2. Code Example for a Simple Component in React Native

State Management with Context API and Redux

State management is crucial in mobile application development to handle the dynamic nature of data and UI state.

- **Context API**: Used for managing local state within smaller applications or specific sections of larger applications, providing a simpler alternative to Redux.

```
import React, { createContext, useContext, useState } from "react";
import { Button, StyleSheet, Text, View } from "react-native";
// Create a Context
const MyContext = createContext();
const MyProvider = ({ children }) => {
  const [value, setValue] = useState("Initial Value");
  return (
    <MyContext.Provider value={{ value, setValue }}>
      {children}
    </MyContext.Provider>
  );
};
const DisplayComponent = () => {
  const { value } = useContext(MyContext);
  return <Text style={styles.text}>{value}</Text>;
};
const UpdateComponent = () => {
  const { setValue } = useContext(MyContext);
  return (
    <Button title="Update Value" onPress={() => setValue("Updated Value")} />
  );
};
const ContextAPIExample = () => {
  return (
    <MyProvider>
      <View style={styles.container}>
        <DisplayComponent />
        <UpdateComponent />
      </View>
    </MyProvider>
  );
};
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
  },
});
export default ContextAPIExample;
```

Figure 3. Code Example for Context API

2.3.1 Creating Dynamic Mobile Screens

Creating dynamic mobile screens enhances interactivity and user engagement. I employed various techniques and tools to achieve this:

- React Navigation: Implemented React Navigation to handle navigation within the application, enabling the creation of multi-screen applications with smooth transitions between screens.

- Dynamic Data Binding: Utilized dynamic data binding to update the UI in real-time as data changes, providing a responsive and interactive user experience.

```
import * as React from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import { View, Text, Button } from 'react-native';
const HomeScreen = ({ navigation }) => {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>Home Screen</Text>
      <Button
        title="Go to Details"
        onPress={() => navigation.navigate('Details')}
      />
    </View>
  );
};
const DetailsScreen = () => {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>Details Screen</Text>
    </View>
  );
};
const Stack = createStackNavigator();
const NavigationExample = () => {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Home">
        <Stack.Screen name="Home" component={HomeScreen} />
        <Stack.Screen name="Details" component={DetailsScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
};
export default NavigationExample;
```

Figure 4. Code Example for React Navigation

2.3.2 Entity Framework Core for Database Management

Entity Framework Core (EF Core) is an Object-Relational Mapping (ORM) framework that facilitates database interactions and data modeling, adhering to the principles of Domain-Driven Design (DDD).

DDD emphasizes a domain-centric approach to software design, focusing on

modeling the business domain and its interactions rather than just data persistence.

2.3.3 Domain-Driven Design (DDD) Architecture

Domain-Driven Design emphasizes the importance of structuring software projects around business domains, focusing on domain logic and modeling rather than technical concerns.

EF Core aligns well with DDD principles by offering robust capabilities that support domain-centric development.

2.3.4 Domain Models and EF Core Entities

Entities: In DDD, entities represent objects with a distinct identity and lifecycle within the domain.

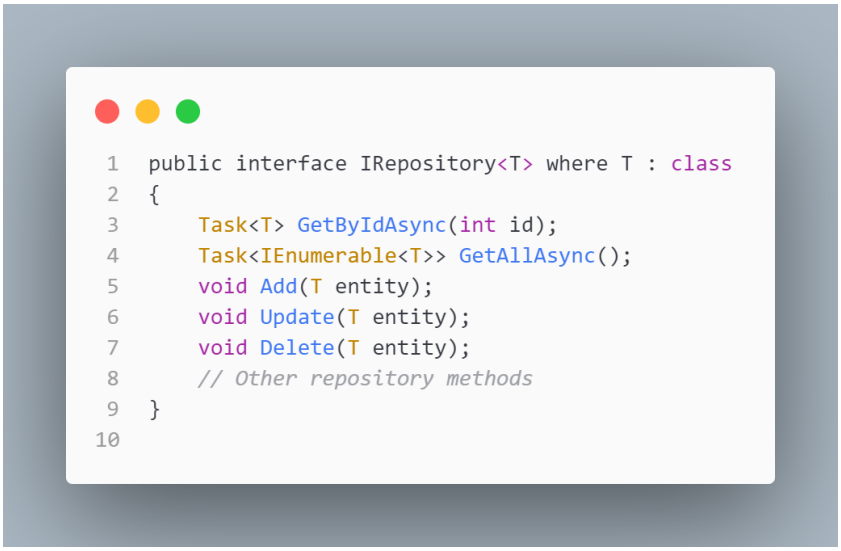
- EF Core allows developers to define entities as classes mapped to database tables.
- For instance, a Product entity in the domain model can be mapped to a Products table in the database.

Figure 5. Model Product

Repository Pattern

Repository Interfaces: Repositories in DDD act as gateways for accessing and managing domain entities.

EF Core facilitates the implementation of the Repository Pattern by providing mechanisms to define repositories that encapsulate data access logic.



```
public interface IRepository<T> where T : class
{
    Task<T> GetByIdAsync(int id);
    Task<IEnumerable<T>> GetAllAsync();
    void Add(T entity);
    void Update(T entity);
    void Delete(T entity);
    // Other repository methods
}
```

Figure 6. Interface for Repository

```
public class ProductRepository : IRepository<Product>
{
    private readonly DbContext _context;
    public ProductRepository(DbContext context)
    {
        _context = context;
    }
    public async Task<Product> GetByIdAsync(int id)
    {
        return await _context.Set<Product>().FindAsync(id);
    }
    // Implementation of other repository methods
}
```

Figure 7. Implement IProductRepository in a ProductRepository class

Database Context

DbContext: EF Core's DbContext class acts as a bridge between the domain model and the database.

It manages database connections, transactions, and serves as the main entry point for querying and saving data.

```
public class AppDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }
    // Other DbSet properties for different entities
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Configurations for entity relationships, indexes, etc.
    }
}
```

Figure 8. Database context sample

Migration Support:

Database Migrations: EF Core's migration feature enables developers to evolve the database schema over time in sync with application changes.

This ensures seamless updates to the database structure without manually altering the database schema.

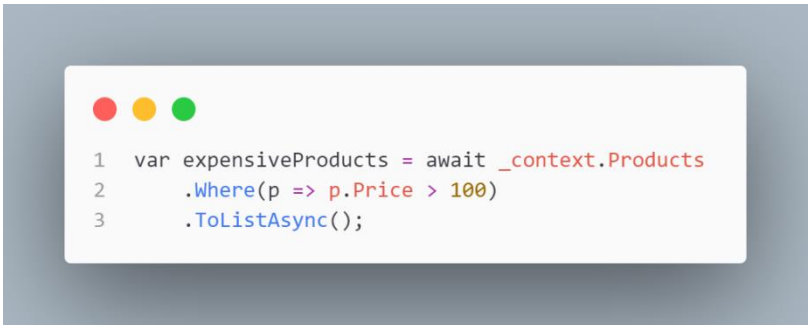
```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

Figure 9. Database migration command

LINQ Integration

Language Integrated Query (LINQ): LINQ in EF Core provides a powerful and type-safe querying mechanism that allows developers to express

queries using familiar C# syntax. This integration enhances developer productivity by reducing the complexity of SQL queries and promoting code readability.



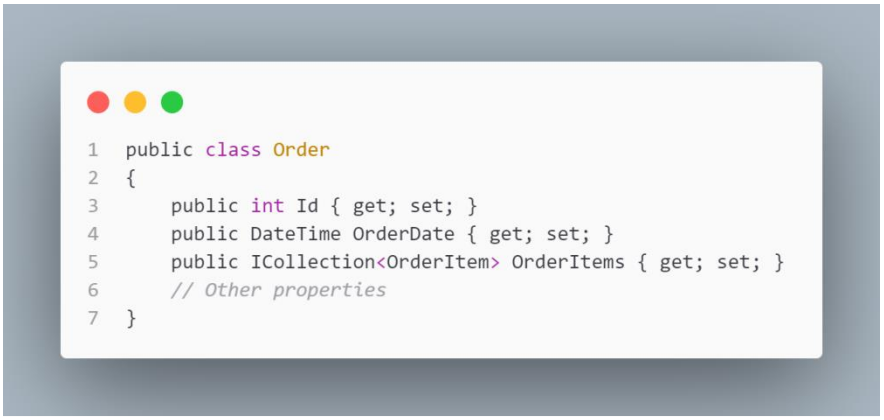
```
var expensiveProducts = await _context.Products
    .Where(p => p.Price > 100)
    .ToListAsync();
```

Figure 10. Sample LINQ command

Aggregate Roots and Relationships

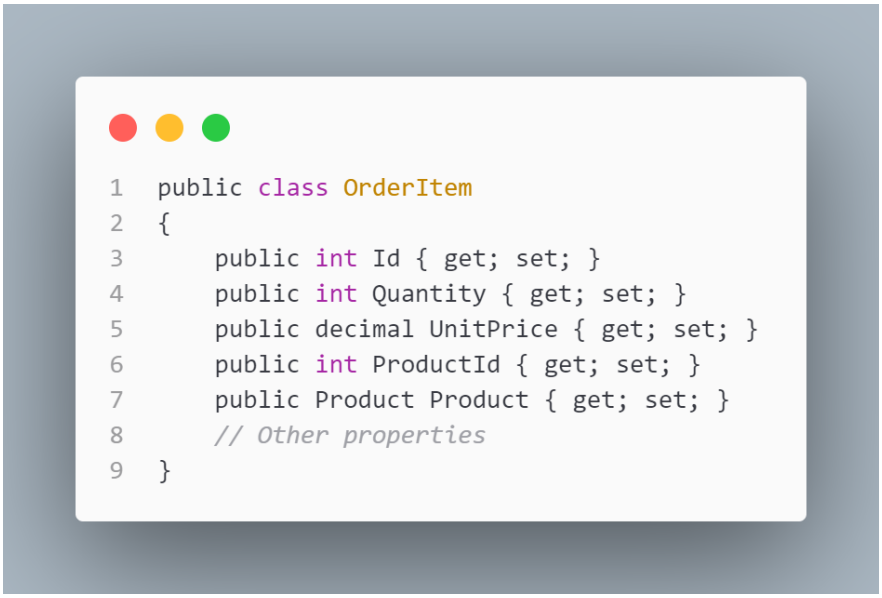
Aggregate Design: In DDD, aggregates are clusters of associated domain objects treated as a single unit.

EF Core supports defining aggregate roots and managing relationships between entities through navigation properties, ensuring consistency and integrity within the domain model.



```
public class Order
{
    public int Id { get; set; }
    public DateTime OrderDate { get; set; }
    public ICollection<OrderItem> OrderItems { get; set; }
    // Other properties
}
```

Figure 11. Order table entities



```
1  public class OrderItem
2  {
3      public int Id { get; set; }
4      public int Quantity { get; set; }
5      public decimal UnitPrice { get; set; }
6      public int ProductId { get; set; }
7      public Product Product { get; set; }
8      // Other properties
9  }
```

Figure 12. OrderItem table entities

Design Patterns: Repository Pattern and Dependency Injection

In modern software development, especially within the ASP.NET Core framework, design patterns like the Repository Pattern and Dependency Injection play crucial roles in promoting code reusability, maintainability, and separation of concerns.

Repository Pattern

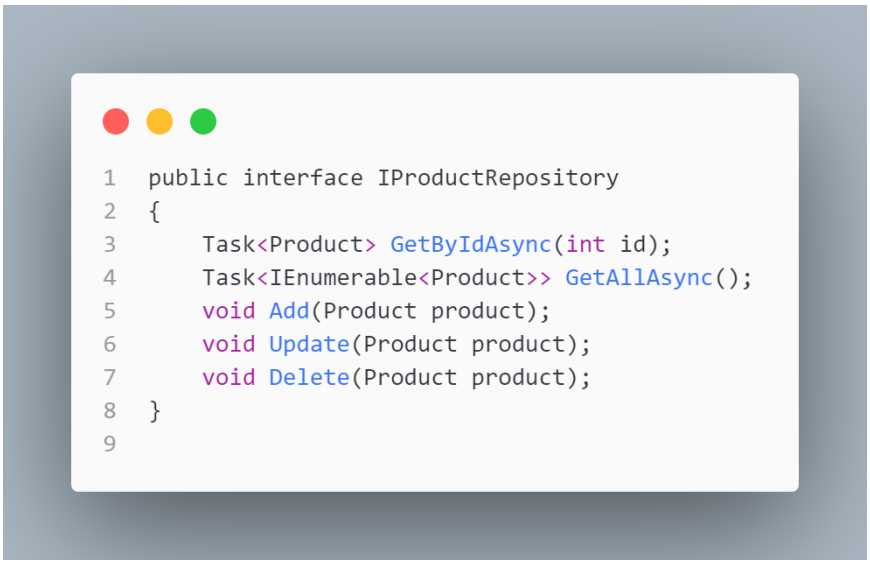
The Repository Pattern is a popular design pattern used to encapsulate the data access logic.

It acts as a mediator between the domain and the data access layers, providing a clean API for querying and persisting domain objects.

The benefit development process:

- **Encapsulation of Data Access Logic**:
    - **Abstraction**: By abstracting the data access logic, the Repository Pattern hides the details of data storage, making it easier to change the underlying data access technology without affecting the business logic.

- **Separation of Concerns**: This pattern helps in separating the data access code from the business logic, enhancing the readability and maintainability of the code.



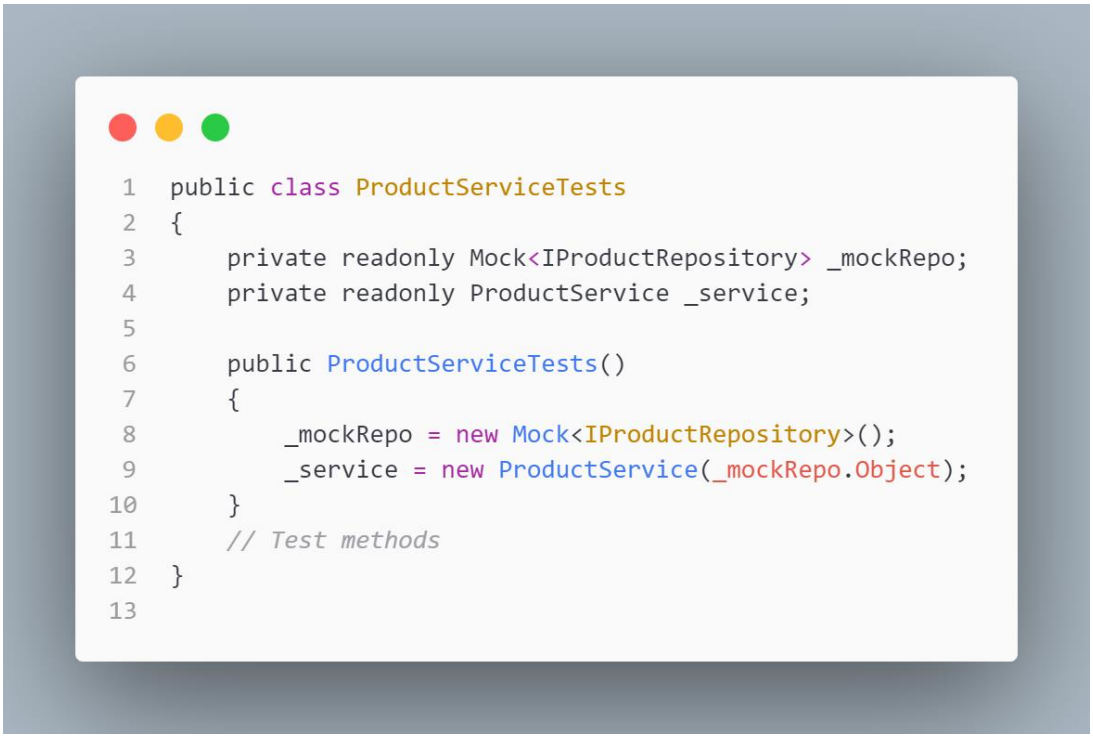
```
public interface IProductRepository
{
    Task<Product> GetByIdAsync(int id);
    Task<IEnumerable<Product>> GetAllAsync();
    void Add(Product product);
    void Update(Product product);
    void Delete(Product product);
}

```

Figure 13. Product interface repository sample

```
public class ProductRepository : IProductRepository
{
    private readonly AppDbContext _context;

    public ProductRepository(AppDbContext context)
    {
        _context = context;
    }
    public async Task<Product> GetByIdAsync(int id)
    {
        return await _context.Products.FindAsync(id);
    }
    public async Task<IEnumerable<Product>> GetAllAsync()
    {
        return await _context.Products.ToListAsync();
    }
    public void Add(Product product)
    {
        _context.Products.Add(product);
        _context.SaveChanges();
    }
    public void Update(Product product)
    {
        _context.Products.Update(product);
        _context.SaveChanges();
    }
    public void Delete(Product product)
    {
        _context.Products.Remove(product);
        _context.SaveChanges();
    }
}
```

Figure 14. Product repository sample

- **Ease of Testing**:
    - **Mocking Repositories**: Since repositories are abstracted through interfaces, it becomes easier to mock them during unit testing, ensuring that tests focus on the business logic without worrying about the actual data access implementation.



```
public class ProductServiceTests
{
    private readonly Mock<IProductRepository> _mockRepo;
    private readonly ProductService _service;

    public ProductServiceTests()
    {
        _mockRepo = new Mock<IProductRepository>();
        _service = new ProductService(_mockRepo.Object);
    }
    // Test methods
}
```

Figure 15. Product service sample

- **Consistency and Reusability**
    - **Reusable Components**: By centralizing data access logic within repositories, these components can be reused across different parts of the application, promoting consistency and reducing redundancy.

Dependency Injection

Dependency Injection (DI) is a design pattern used to achieve Inversion of Control (IoC) between classes and their dependencies.

ASP.NET Core provides a built-in Dependency Injection container, making it a core part of the framework. Here's how DI enhances application design:

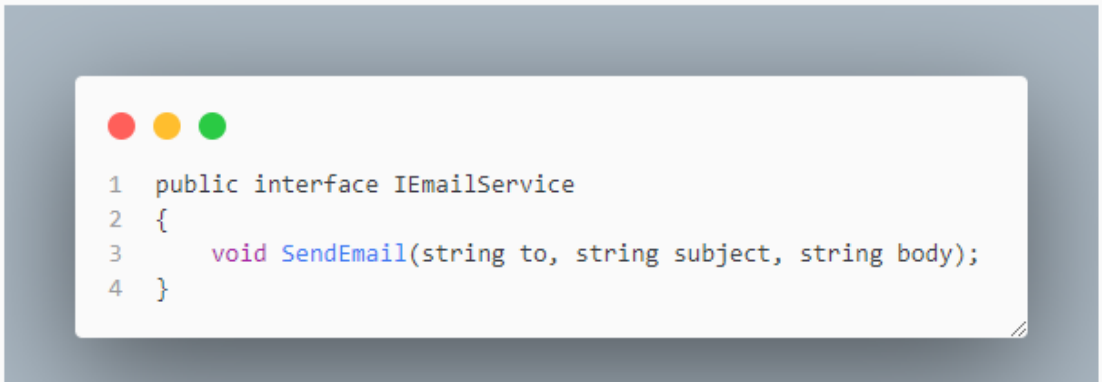
- **Loose Coupling**:
    - **Decoupling Dependencies**: DI promotes loose coupling by allowing the dependencies of a class to be injected rather than being hardcoded. This decoupling makes it easier to manage and

modify dependencies.

```
public class ProductService
{
    private readonly IProductRepository _productRepository;
    public ProductService(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }
    // Service methods
}
```

Figure 16. Product service sample

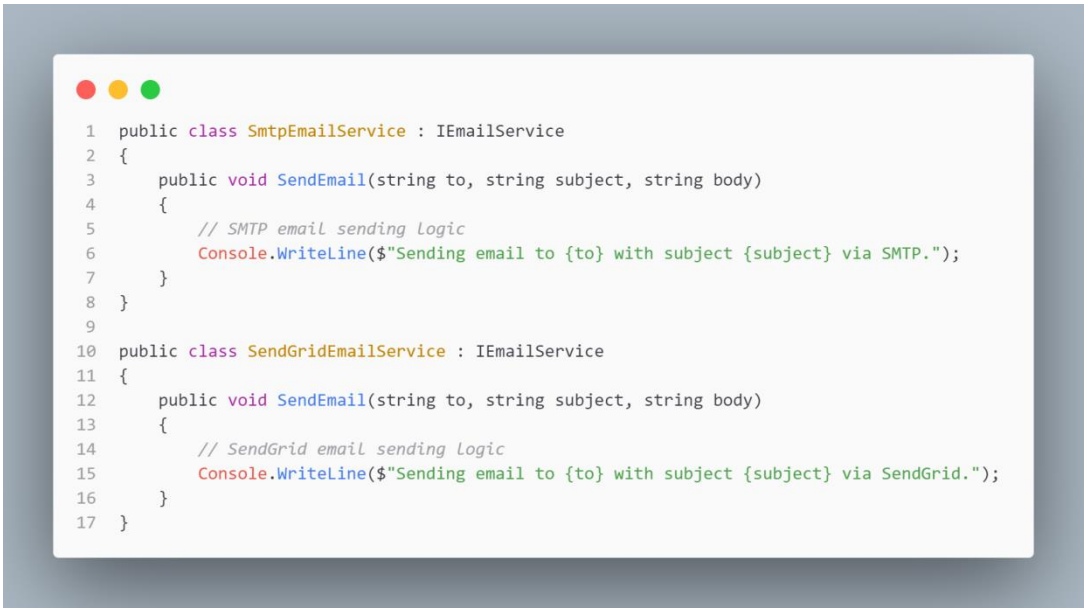
- **Modular Development**:
    - Dependency Injection (DI) supports the creation of modular components that can be easily replaced or updated without affecting other parts of the application.
    - This modularity enhances code maintainability and extensibility. Here's an example to demonstrate modular development and component reusability in ASP.NET Core:
        - *Step 1: Define the Service Interface*



```
public interface IEmailService
{
    void SendEmail(string to, string subject, string body);
}
```

Figure 17. IEmailServer interface sample

- Step 2: Implement the Service



```
public class SmtpEmailService : IEmailService
{
    public void SendEmail(string to, string subject, string body)
    {
        // SMTP email sending logic
        Console.WriteLine($"Sending email to {to} with subject {subject} via SMTP.");
    }
}

public class SendGridEmailService : IEmailService
{
    public void SendEmail(string to, string subject, string body)
    {
        // SendGrid email sending logic
        Console.WriteLine($"Sending email to {to} with subject {subject} via SendGrid.");
    }
}
```

Figure 18. Email Server sample

- Step 3: Register Services in the Service Container

```
public void ConfigureServices(IServiceCollection services)
{
    // Register SmtpEmailService for IEmailService
    services.AddScoped<IEmailService, SmtpEmailService>();

    // Alternatively, you can use SendGridEmailService
    // services.AddScoped<IEmailService, SendGridEmailService>();

    services.AddControllers();
}
```

Figure 19. Configure Services sample

- Step 4: Use the Service in a Controller

```
[ApiController]
[Route("api/[controller]")]
public class NotificationController : ControllerBase
{
    private readonly IEmailService _emailService;

    public NotificationController(IEmailService emailService)
    {
        _emailService = emailService;
    }

    [HttpPost("send-email")]
    public IActionResult SendEmail(string to, string subject, string body)
    {
        _emailService.SendEmail(to, subject, body);
        return Ok("Email sent successfully.");
    }
}
```

Figure 20. Notification controller sample

- Step 5: Easily Replace or Update Components
    - By simply changing the service registration in ConfigureServices, we can switch between different email service implementations without modifying the controller or other parts of the application.
    - For example, to switch from SmtpEmailService to SendGridEmailService:

```
1  public void ConfigureServices(IServiceCollection services)
2  {
3      // Use SendGridEmailService instead of SmtpEmailService
4      services.AddScoped<IEmailService, SendGridEmailService>();
5
6      services.AddControllers();
7  }
8
```

Figure 21. Configure Services sample

Enhanced Testability:

Mocking Dependencies: With DI, dependencies are injected, making it straightforward to replace real implementations with mocks during testing. This improves the ability to unit test classes in isolation.

```
1  public class ProductServiceTests
2  {
3      private readonly Mock<IProductRepository> _mockRepo;
4      private readonly ProductService _service;
5
6      public ProductServiceTests()
7      {
8          _mockRepo = new Mock<IProductRepository>();
9          _service = new ProductService(_mockRepo.Object);
10     }
11     [Fact]
12     public async Task GetProductById_ReturnsProduct()
13     {
14         // Arrange
15         var productId = 1;
16         _mockRepo.Setup(repo => repo.GetByIdAsync(productId)).ReturnsAsync(new Product { Id = productId });
17         // Act
18         var result = await _service.GetProductByIdAsync(productId);
19         // Assert
20         Assert.Equal(productId, result.Id);
21     }
22 }
```

Figure 22. Mocking dependence

2.4 Frontend Development with React

Technologies and Tools Used

React JS and React TS

React is a renowned JavaScript library developed by Facebook that facilitates the creation of efficient and flexible user interfaces.

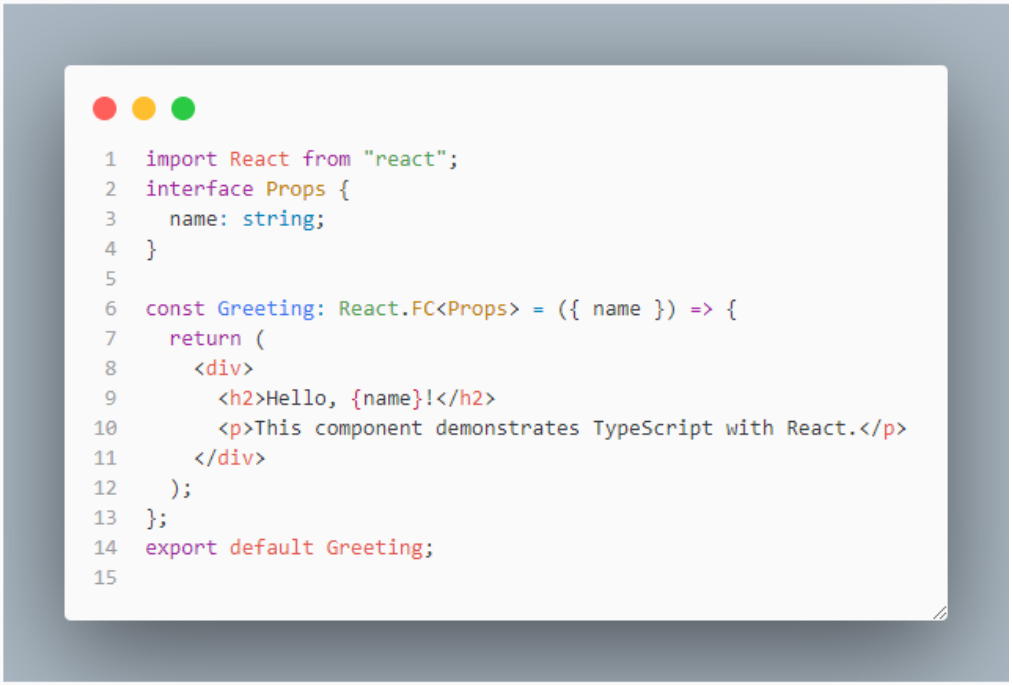
Throughout my work, I utilized both React JS (JavaScript) and React TS (TypeScript) to develop UI components.

- **React JS**: This allowed for the creation of simple and reusable components. React JS makes UI management easier through its component-based approach.

```
1  import React from 'react';
2
3  const HelloWorld = () => {
4    return (
5      <div>
6        <h1>Hello, World!</h1>
7        <p>This is a simple React component.</p>
8      </div>
9    );
10 }
11
12 export default HelloWorld;
13
```

Figure 23. Hello World by react components

- **React TS**: Using TypeScript enhanced code accuracy by adding type checking. This minimizes runtime errors and improves code maintainability.



```tsx
import React from "react";
interface Props {
  name: string;
}

const Greeting: React.FC<Props> = ({ name }) => {
  return (
    <div>
      <h2>Hello, {name}!</h2>
      <p>This component demonstrates TypeScript with React.</p>
    </div>
  );
};
export default Greeting;

```

Figure 24. Greeting typescript components

Redux and State Management

Redux is a widely-used state management library in React applications.

Redux helps manage application state in a centralized and straightforward manner, especially as the application scales.

- **State Management**: With Redux, I could manage the global state of the application, allowing components to consistently access and update the state.
- **Redux Thunk**: Using middleware like Redux Thunk for handling asynchronous tasks, such as API calls, made the code clearer and more comprehensible.

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers'; // assuming you have a combined reducer

const store = createStore(
  rootReducer,
  applyMiddleware(thunk)
);

export default store;
```

Figure 25. Redux-thunk sample

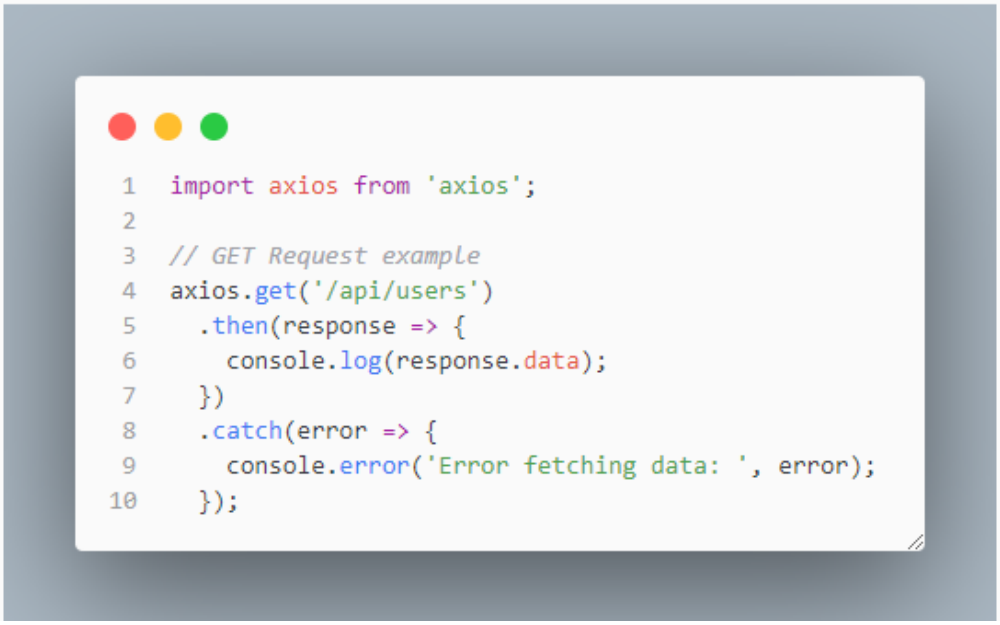
```
import axios from 'axios';

export const fetchData = () => async (dispatch) => {
  try {
    const response = await axios.get('/api/data');
    dispatch({ type: 'FETCH_DATA_SUCCESS', payload: response.data });
  } catch (error) {
    dispatch({ type: 'FETCH_DATA_FAILURE', error: error.message });
  }
};
```

Figure 26. Dispatch data with Redux sample

Axios and HTTP Requests

Axios is a powerful HTTP client library that simplifies sending HTTP requests to backend APIs efficiently.

- **GET Requests**: I used Axios to send GET requests, retrieve data from the server, and update the UI in real-time.



```
import axios from 'axios';

// GET Request example
axios.get('/api/users')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Error fetching data: ', error);
  });
```

Figure 27. Axios get request sample

POST Requests: I used Axios to send data from the client to the server, supporting functions like registration, login, and other CRUD operations.

```
// POST Request example
const newUser = { username: 'newuser', password: 'password123' };
axios.post('/api/users', newUser)
  .then(response => {
    console.log('User created:', response.data);
  })
  .catch(error => {
    console.error('Error creating user: ', error);
  });
```

Figure 28. Axios post request sample

Performance Optimization

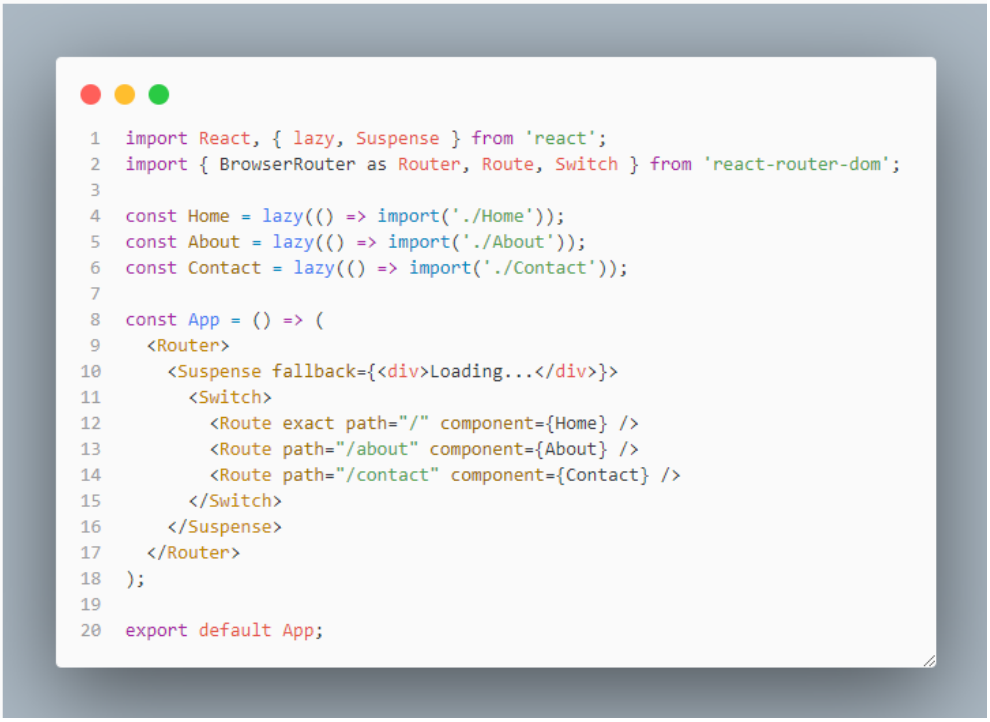
Web application performance is crucial for enhancing user experience. During development, I applied optimization techniques such as Code Splitting and Lazy Loading.

- **Code Splitting**: By using Code Splitting, I broke down JavaScript files into smaller chunks, reducing initial load times and improving overall performance.

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

const App = () => (
  <div>
    <Suspense fallback={<div>Loading...</div>}>
      <OtherComponent />
    </Suspense>
  </div>
);

export default App;
```

Figure 29. Code Splitting with React sample

- **Lazy Loading**: I implemented Lazy Loading to load only necessary components when needed by the user, minimizing initial load size and speeding up page load times.



```jsx
import React, { lazy, Suspense } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

const Home = lazy(() => import('./Home'));
const About = lazy(() => import('./About'));
const Contact = lazy(() => import('./Contact'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route path="/contact" component={Contact} />
      </Switch>
    </Suspense>
  </Router>
);

export default App;
```

Figure 30. React Lazy Loading sample

UI Components and Dynamic Web Pages

Designing and Implementing UI Components

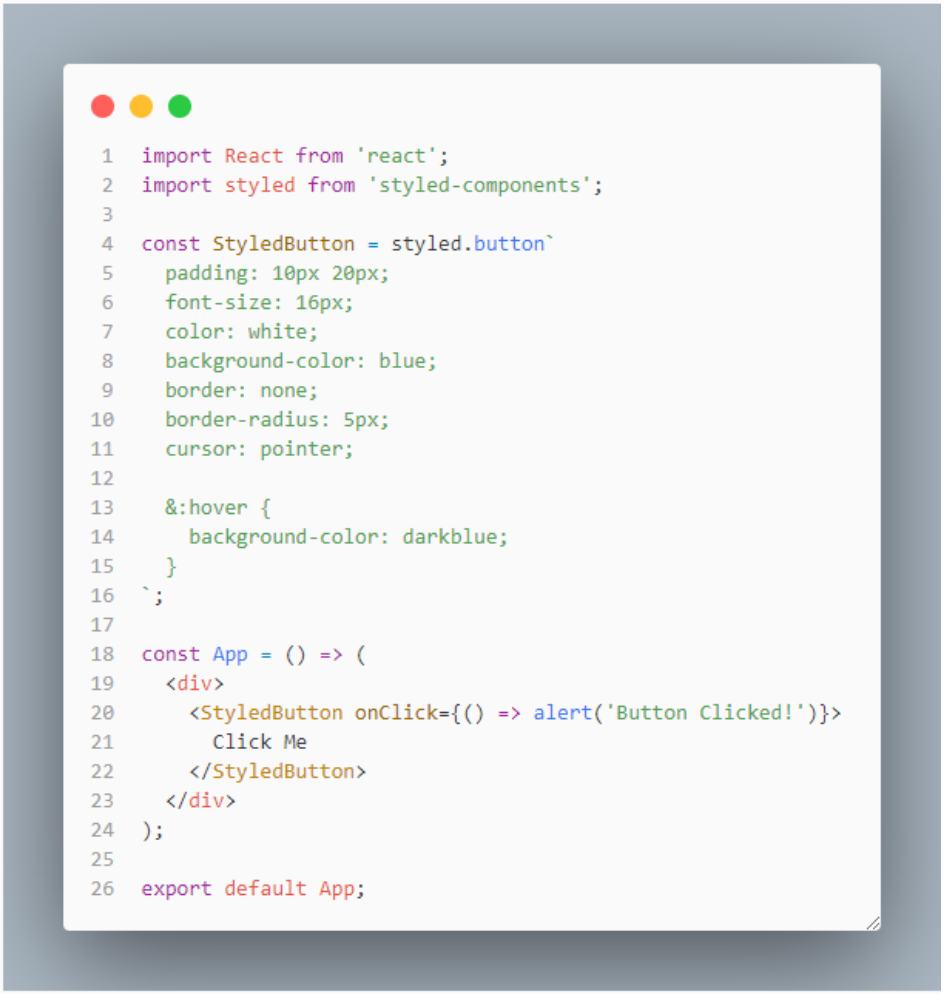
Designing and implementing UI components is a critical aspect of frontend development. I created simple, user-friendly, and reusable UI components.

- **Component-based Architecture**: I built UI components based on a component structure, making the code modular and maintainable.

```jsx
import React from 'react';

const Button = ({ onClick, children }) => (
  <button onClick={onClick} style={{ padding: '10px 20px', fontSize: '16px' }}>
    {children}
  </button>
);

export default Button;
```

Figure 31. Button component with React sample

Styled Components: I used the Styled Components library to write CSS directly within JavaScript, effectively managing styles.



```
import React from 'react';
import styled from 'styled-components';

const StyledButton = styled.button`
  padding: 10px 20px;
  font-size: 16px;
  color: white;
  background-color: blue;
  border: none;
  border-radius: 5px;
  cursor: pointer;

  &:hover {
    background-color: darkblue;
  }
`;

const App = () => (
  <div>
    <StyledButton onClick={() => alert('Button Clicked!')}>
      Click Me
    </StyledButton>
  </div>
);

export default App;
```

Figure 32. Styled components sample with React

Creating Dynamic Web Pages

Creating dynamic web pages enhances interactivity and user experience.

I employed the following techniques and tools:

- **React Router**: I used React Router to handle navigation within the application, enabling the creation of multi-page websites with smooth transitions between pages.

```jsx
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Home from './Home';
import About from './About';
import Contact from './Contact';

const App = () => {
  return (
    <Router>
      <div>
        <Switch>
          <Route exact path="/" component={Home} />
          <Route path="/about" component={About} />
          <Route path="/contact" component={Contact} />
        </Switch>
      </div>
    </Router>
  );
}

export default App;
```

Figure 33. React Router components sample

- **Dynamic Data Binding**: I used dynamic data binding techniques to update the UI in real-time as data changes.

```jsx
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const DataDisplay = () => {
  const [data, setData] = useState([]);

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios.get('/api/data');
      setData(result.data);
    };
    fetchData();
  }, []);

  return (
    <div>
      <h1>Data List</h1>
      <ul>
        {data.map(item => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default DataDisplay;
```

Figure 34. Dynamic data binding with React sample

CHAPTER 3. Project And Knowledge

3.1 Introduction

Overview

The project involves developing two main systems for a retail business:

- The Admin Web System and the POS (Point of Sale) Web App.
- The Admin Web System allows store administrators to manage store information, products, materials, and other critical data.
- The POS Web App facilitates the creation and processing of orders.
- Both systems are built using React TypeScript for the front end, .NET Core API for the back end, and SQL Server as the database.

Scope

- **Admin Web System**:
    1. This system provides a user interface for store administrators to manage store details, products, and other resources.
    2. It includes user authentication and comprehensive data management capabilities.

3.2 System Architecture

3.2.1 High-Level Architecture

React Native:

- React Native allows for the development of cross-platform mobile apps using JavaScript and React.
- It leverages native platform components to render UIs, which ensures high performance and a native look-and-feel.
- The communication between JavaScript code and the native environment is handled by a bridge that asynchronously interacts with native APIs.
- Important: React Native can build app for Android and iOS platform.

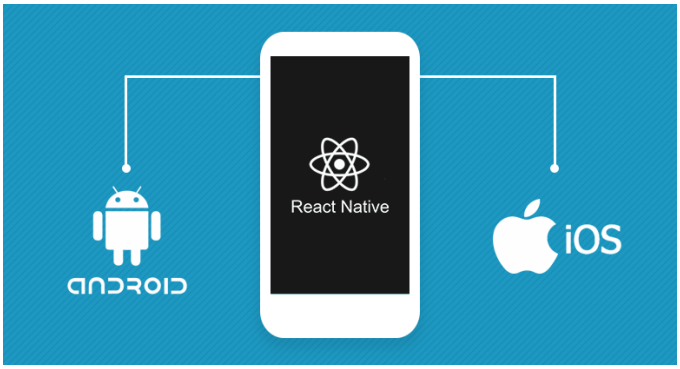


Figure 35. React native and iOS/Android platform

Real-time Communication with SocketIO and SignalR:

- SocketIO and SignalR provide frameworks for adding real-time web functionality to applications, facilitating bidirectional communication between clients and servers.
- By Notification need real-time message so SignalR is a best choice for this case.

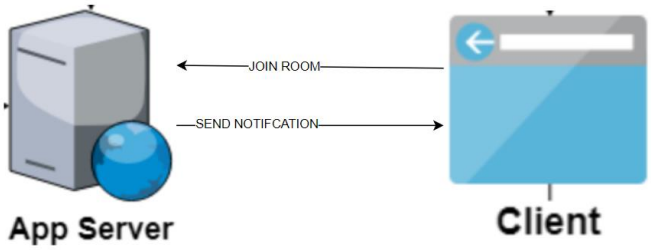


Figure 36. Realtime connection React native Client and Server

Firebase Notifications for React Native:

- Firebase Cloud Messaging (FCM) integrates with mobile and web applications to send push notifications.
- In React Native, libraries like *react-native-firebase* bridge Firebase capabilities with mobile devices.

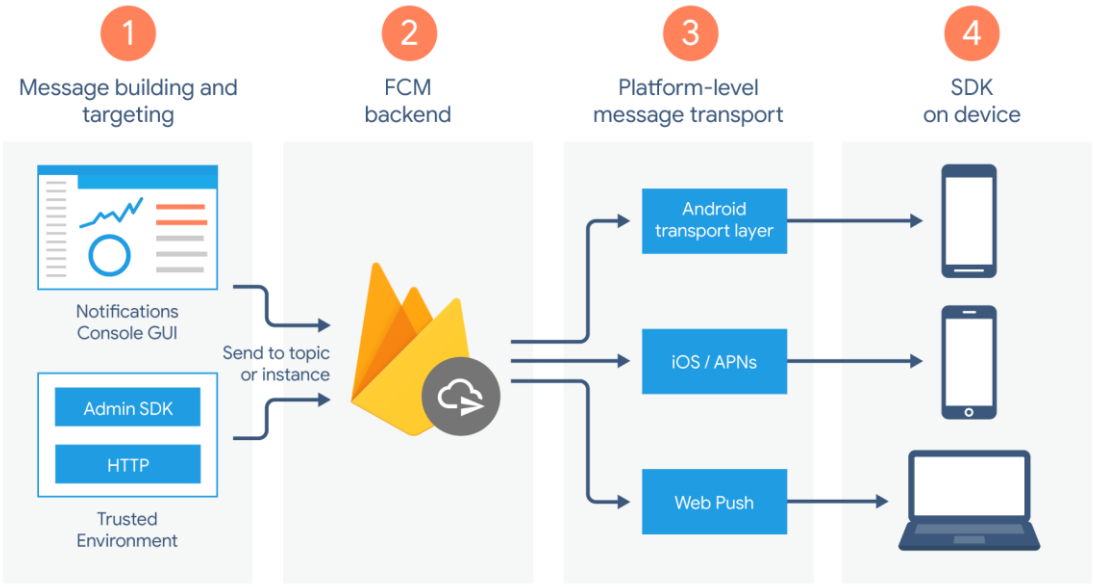


Figure 37. Firebase Cloud Messaging (FCM) architecture for sending notifications

Firebase Cloud Messaging (FCM) architecture for sending notifications to different devices. Description of the model messages are created, processed, and delivered to various platforms using FCM:

- **Message Building and Targeting**:
  - **Notifications Console GUI**: A graphical user interface where messages are created and targeted.
  - **Admin SDK / HTTP**: Trusted environments, such as server-side applications, use the Admin SDK or HTTP to send messages.
  - Messages are sent to a specific topic or instance (device).
- **FCM Backend**:
  - This is the central component of FCM that handles message processing and routing.
- **Platform-Level Message Transport**:
  - **Android Transport Layer**: Responsible for delivering messages to Android devices.

- **iOS**: Uses Apple Push Notification Service (APNs) to deliver messages to iOS devices.
- **Web Push**: Delivers messages to web applications.

- **SDK on Device**:
  - **Android Devices**: Receive messages through the Android transport layer.
  - **iOS Devices**: Receive messages via APNs.
  - **Web Applications**: Receive messages through web push services.

Microservices Architecture & RabbitMQ with Firebase Cloud Messaging Server:

- Microservices architecture involves developing a suite of small, independent services that run each application process as a service.
- These services communicate over a network, often using HTTP REST, or asynchronous messaging systems like RabbitMQ.

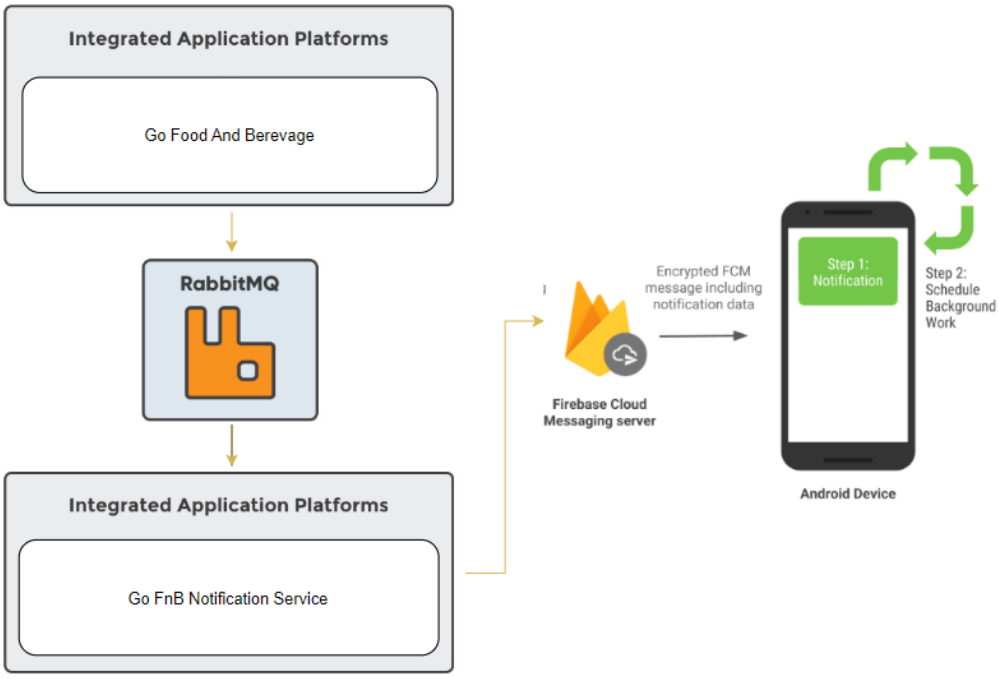


Figure 38. Microservice, RabbitMQ and FCM

The flow of notification data from integrated application platforms through RabbitMQ to Firebase Cloud Messaging (FCM) and then to an Android device.

The end-to-end process of how notification data is generated, processed, and delivered to an Android device using RabbitMQ and Firebase Cloud Messaging:

- **Integrated Application Platforms**:
    - The process begins with an integrated application platform labeled "Go Food And Beverage."
    - This platform generates data that needs to be sent as notifications.
- **RabbitMQ**:
    - The data from the "Go Food And Beverage" platform is sent to RabbitMQ, which acts as a message broker.
    - RabbitMQ handles the queuing and routing of messages.
- **Integrated Application Platforms**:
    - From RabbitMQ, the data is forwarded to another integrated application platform "Go FnB Notification Service."
    - This platform will subscribe message from RabbitMQ and preparing and sending notifications.
- **Firebase Cloud Messaging Server**:
    - The "Go FnB Notification Service" sends an encrypted FCM message, including notification data, to the Firebase Cloud Messaging server.
    - FCM handles the delivery of these messages to the target devices (FirebaseId is a key).
- **Android Device**:
    - The Android device receives the FCM message in two steps:
    - **Step 1: Notification**: The device displays the notification.
    - **Step 2: Schedule Background Work**: The device schedules any required background work based on the notification data.

The high-level architecture of the project is designed to ensure modularity, scalability, and security. The architecture consists of the following layers:

- **Front-End Layer**:
    - Developed using React and TypeScript, this layer provides a responsive and interactive user interface for both the Admin Web System and the POS Web App.
- **Back-End Layer**:
    - Implemented using .NET Core API, this layer handles business logic, data processing, and communication with the database.
- **Database Layer**:
    - SQL Server is used to store and manage all data related to the store, products, users, and transactions.

Components

- **Front-end**: React TypeScript for creating user interfaces.
- **Back-end**: .NET Core API for handling business logic and data processing.
- **Database**: SQL Server for data storage.
- **Security**: Implementing authentication, authorization, and password hashing.
- **API Documentation**: Using NSwag for automatic API documentation generation.

3.2.2 Technology Stack

3.2.2.1 React Native

- **Languages:** JavaScript or TypeScript for coding, along with HTML and CSS for styling.
- **Core Framework:** React is used to define UI components and manage their lifecycle.

- **State Management:** Redux or Context API can be used to manage the app's state across different components.
- **Navigation:** React Navigation provides a way for your app to transition between screens where each new screen is placed on top of a stack.

Example: In a React Native app, I might have a UserProfile screen. Using React Navigation, I can navigate to this screen with a button press, passing parameters like user ID. The state management through Redux can help maintain user data throughout the app.

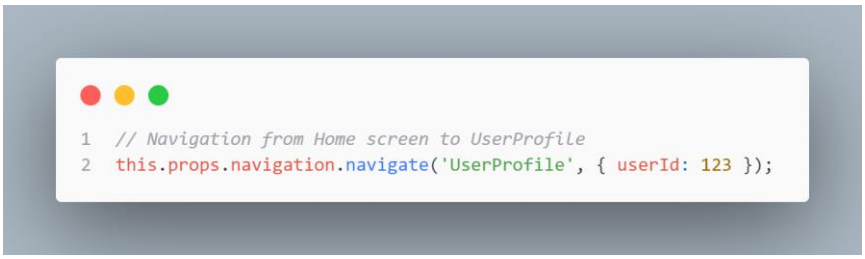


Figure 39. React native navigate example.

3.2.2.2 Real-time Communication with SocketIO and SignalR

- **SocketIO:**
  - **Server:** Node.js with express
  - **Client:** Can be any web client (JavaScript), or native platforms
  - **Fallbacks:** WebSockets, AJAX long polling
- **SignalR:**
  - **Server:** ASP.NET Core
  - **Client:** Any client that supports HTTP requests
  - **Protocols:** WebSockets, Server-Sent Events, Long Polling

Example: A chat application using Socket.IO might emit a message from one client and broadcast it to others in real-time.

```
// Server-side: Emitting an event
io.on('connection', (socket) => {
    socket.on('sendMessage', (message) => {
      io.emit('receiveMessage', message);
    });
  });
```

Figure 40. Emit data to Socket server at React native.

```
<SignalRContext.Provider
            appState={appState}
            connectEnabled={branchId !== '' && deviceInformation != null}
            invokeJoinRoom={{BranchId: branchId, Details: deviceInformation}}
            dependencies={[branchId, isOnline]}
            onReconnect={(connection: any) => {
              if (connection) {
                notificationDispatch.initialize(branchId);
              }
            }}
            onJoinRoom={(connection: any) => {
              if (connection) {
                notificationDispatch.initialize(branchId);
              }
            }}
            url={NOTIFICATION_SERVER_URL}>
            {isReady && <AppProvider>{Content}</AppProvider>}
            <Toast config={isTablet ? customToastConfigs : customToastConfigsMobile} />
            {ContentConfirm}
          </SignalRContext.Provider>
```

Figure 41. React native app SignalR example.

This code below is part of an application that likely involves real-time communication, possibly using SignalR, where devices join a room or group based on their connection details.

The method ensures that the device information is sent and the user is added to the appropriate group if the necessary connection details are provided.

```
public async Task JoinRoom(JoinRoomDeviceDto userConnection)
        {
            if (!string.IsNullOrEmpty(userConnection.BranchId) && userConnection?.Details?.DeviceId != null)
            {
                var userDevice = new DeviceDto
                {
                    FirebaseId = userConnection.Details.FirebaseId ?? string.Empty,
                    StoreId = userConnection.Details.StoreId,
                    BranchId = userConnection.BranchId,
                    DeviceId = userConnection.Details.DeviceId ?? string.Empty,
                    Role = userConnection.Details.Role,
                    PlatformId = userConnection.Details.PlatformId,
                    PlatformName = userConnection.Details.PlatformName ?? string.Empty,
                    PlatformVersion = userConnection.Details.PlatformVersion ?? string.Empty,
                    Language = userConnection.Details.Language,
                };

                await SendDeviceInfo(userDevice, Context.ConnectionId);
                await Groups.AddToGroupAsync(Context.ConnectionId, $"{userConnection.BranchId}");
            }
            else
            {
                _logger.Error("Cannot find Branch and DeviceId");
            }
        }
```

Figure 42. SignalR on Back-End Server .Net

3.2.2.3 Firebase Notifications for React Native

- **Backend Service:** Firebase Cloud Messaging
- **Client-side Library:** react-native-firebase for integration
- **Notifications:** Supports both foreground and background notifications, as well as data messages.

```
messaging().setBackgroundMessageHandler(async remoteMessage => {
  const {data} = remoteMessage;
  const {branchId, detail} = data;
  const now = Date.now().toString();
  const message = JSON.parse(detail);
  const notification = {
    id: now,
    type: EnumNotificationType.ORDER,
    branchId: branchId,
    subtype: EnumNotificationSubtype.ORDER_CONFIRM,
    createdTime: now,
    orderCode: message.orderCode ?? message.StringCode,
    orderId: message.orderId ?? message.OrderId,
  };
  notificationListDispatch.append(notification, EnumNotificationStatus.UNREAD, branchId);
});
```

Figure 43. Take message from FCM on React Native.

This code below is part of an application that handles sending push notifications via Firebase Cloud Messaging (FCM).

It initializes the Firebase app, processes the messages in batches, and sends each batch asynchronously. The use of batch processing helps manage large lists of notifications efficiently.

```
public async Task SendMessageAsync(SendMessage message, CancellationToken cancellationToken = default)
        {
            try
            {
                FirebaseApp firebaseAdmin = FirebaseApp.GetInstance(message.BranchId);
                if (firebaseAdmin == null)
                {
                    FirebaseApp.Create(new AppOptions()
                    {
                        Credential = message.Credential,
                    }, message.BranchId);
                    firebaseAdmin = FirebaseApp.GetInstance(message.BranchId);
                }

                if (firebaseAdmin != null)
                {
                    var batchSize = DefaultConstants.DEFAULT_MAX_PUSH_NOTIFICATION_FCM;
                    var listFirebaseCount = message?.ListFirebase?.Count() ?? 0;
                    var numberOfBatches = Math.Round((double)listFirebaseCount / batchSize);

                    for (var i = 0; i <= numberOfBatches; i++)
                    {
                        string[] dataPush;
                        if (message.ListFirebase != null && message.ListFirebase.Count() > i * batchSize)
                        {
                            int start = i * batchSize;
                            int end = Math.Min((i + 1) * batchSize, message.ListFirebase.Count());
                            dataPush = message.ListFirebase.Skip(start).Take(end - start).ToArray();
                        }
                        else
                        {
                            dataPush = [];
                        }

                        if (dataPush.Length > 0)
                        {
                            await PushNotificationAsync(firebaseAdmin, dataPush, message);
                        }
                    }
                }
            }
            catch (Exception ex)
            {
                _logger.Error($"Error on ${nameof(SendMessageAsync)}", ex);
            }
        }
```

Figure 44. Send message from .net server to FCM

3.2.2.4 Microservices Architecture & RabbitMQ

Microservice is an architectural style that structures an application as a collection of small, autonomous services modeled around a business domain.

Each microservice is a self-contained unit that encapsulates a specific piece of functionality and communicates with other microservices through well-defined APIs.

Key characteristics of microservices:

- **Single Responsibility**: Each microservice is designed to perform a specific business function and does it well.

- **Independence**: Microservices are independently deployable and scalable. Each service can be developed, deployed, and scaled without affecting other services.
- **Decentralized Data Management**: Each microservice manages its own database or data storage, ensuring loose coupling and independent evolution.
- **Technology Diversity**: Different microservices can be built using different technologies, languages, or frameworks, allowing teams to choose the best tools for the job.
- **Inter-Service Communication**: Microservices communicate with each other using lightweight protocols such as HTTP/REST, g-RPC, or messaging queues.

Benefits of microservices:

- **Scalability**: Services can be scaled independently based on their load, improving overall system performance and resource utilization.
- **Resilience**: Failure of one service does not necessarily bring down the entire system. Microservices can be designed to handle failure gracefully and provide fault isolation.
- **Flexibility in Development**: Teams can develop, deploy, and scale microservices independently, leading to faster development cycles and continuous delivery.
- **Maintainability**: Smaller codebases are easier to understand, maintain, and test. Each microservice can be updated and deployed independently.
- **Technological Freedom**: Teams can use different technologies and frameworks for different services, allowing them to leverage the best tools for specific tasks.

Message Broker:

RabbitMQ handles inter-service messages, decoupling services from each other. RabbitMQ is a popular message broker that facilitates communication between microservices.

Beneficial in a microservices architecture for several reasons:

- **Decoupling**:
    - **Loose Coupling**: RabbitMQ enables loose coupling between services, meaning services can communicate without needing to know each other's details or even being aware of each other’s existence. This decoupling makes the system more resilient and easier to maintain.
- **Asynchronous Communication**:
    - **Non-Blocking**: Services can send messages to RabbitMQ without waiting for an immediate response, allowing them to continue processing other tasks. This is useful for tasks that can be processed in the background or in parallel.
- **Scalability**:
    - **Distributed Processing**: RabbitMQ can distribute messages across multiple consumers, which allows the system to scale horizontally by adding more consumers to handle increased load.
- **Reliability**:
    - **Durable Messaging**: RabbitMQ supports message durability, ensuring messages are not lost in case of broker failures. This is crucial for maintaining data integrity and ensuring reliable communication between services.
- **Load Balancing**:
    - **Message Routing**: RabbitMQ can route messages to different consumers based on routing keys, which can be used to balance the load among multiple instances of a service.
- **Fault Tolerance**:

- **Acknowledgments and Retries**: RabbitMQ provides message acknowledgment mechanisms to ensure messages are processed successfully. In case of failures, messages can be re-queued and retried.

- **Flexibility**:
  - **Multiple Protocols**: RabbitMQ supports various messaging protocols, including AMQP, MQTT, and STOMP, making it versatile for different types of communication needs.
  - **Pluggable Architecture**: RabbitMQ has a modular design that allows for the integration of plugins to extend its functionality.
- **Transactional Messaging**:
  - **Atomic Operations**: RabbitMQ supports transactions, which allows you to publish and consume messages within a single transaction, ensuring consistency.

Two services, Order Service communicate via RabbitMQ. When an order is placed, the Order Service publishes a message to a queue that the POS Notification Service is subscribed to.

After subscribing to the message from RabbitMQ, the system will send a notification signal to the react native application via SignalR or FCM.

```
/// <summary>
/// Sends a POS notification to a specific branch.
/// <para>IMPORTANT: "Payload" The content of the POS notification, must be a valid JSON string.</para>
/// </summary>
/// <returns>Task.</returns>
public async Task SendPOSNotificationAsync(Guid branchId, string notificationType, string payload, CancellationToken cancellationToken = default)
{
    var notificationEvent = new POSNotificationEvent(branchId, notificationType, payload);
    await _publishEndpoint.Publish<POSNotificationEvent>(notificationEvent, cancellationToken);
}
```

Figure 45. Publish message to RabbitMQ server.

```
public override async Task Consume(ConsumeContext<POSNotificationEvent> context)
    {
        try
        {
            _logger.Information("Consume Message POS Notification:", context.Message.Payload.ToString());
            Guid branchId = context.Message.BranchId;
            string typeNotification = context.Message.TypeNotification;
            string payload = context.Message.Payload;
            switch (typeNotification)
            {
                case NotificationListenerConstants.POS_NOTIFY_NEW_ORDER_FROM_APP_TO_POS:
                    await _posNotificationService.SendNotificationSignalAndFirebaseToClientAsync(branchId, typeNotification, payload);
                    break;
                case NotificationListenerConstants.AUTO_END_SHIFT:
                    await _deviceService.UpdateDeviceDeActiveEndShiftAsync();
                    break;
                default:
                    await _posNotificationService.SendNotificationSignalToClientAsync(branchId, typeNotification, payload);
                    break;
            }
        }
        catch (Exception ex)
        {
            _logger.Error("Error Message POS Notification", ex);
            throw;
        }
    }
```

Figure 46. Consume message from RabbitMQ server.

3.2.2.5 Integration with Momo payment system

First of all, MoMo e-wallet application is a product of Online Mobile Services Joint Stock Company (M_Service). The application allows customer to create and top up with MoMo account to pay for more than 200 services such as phone top-up, water and electricity payments, consumer loan payments, etc…

MoMo service can help customers pay for orders created from the Food And Beverage system quickly and conveniently.

Activity diagram about MoMo service interact with system Food And Beverage:

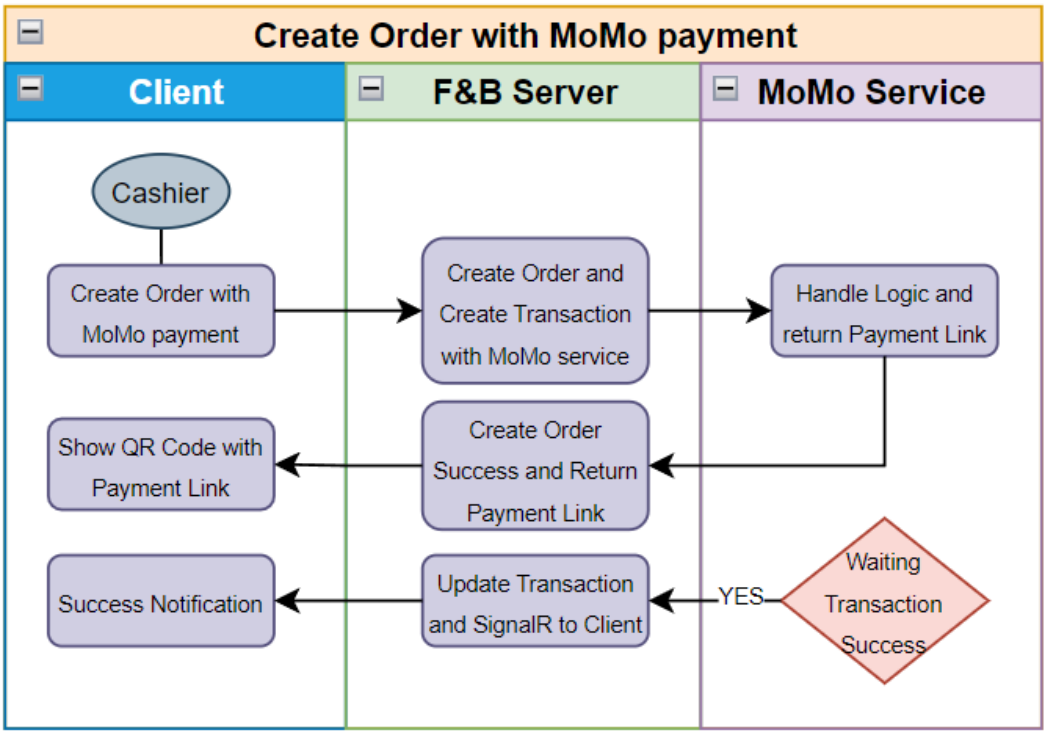


Figure 47. Activity diagram about MoMo service.

- Example:

```
private async Task<CreateGetwayRequestModel> CreateOrderPaymentTransactionAsync(
    CreateMoMoPaymentRequest request,
    Guid? accountId,
    string storeName,
    string branchName,
    string ipnUrl,
    string email,
    bool? isPlatformPOS = false)
{
    var orderInfo = $"Payment for order {request?.OrderCode} at {storeName} - {branchName}";
    //Handle Create OrderPaymentTransaction
    var extractDataJson = new JObject
        {
            { "orderId", request.OrderId.ToString() },
            { "amount", request.Amount },
            { "isPlatformPOS", isPlatformPOS }
        };
    var extractData = Base64Encode(extractDataJson.ToString());
    var orderPaymentTransaction = new OrderPaymentTransaction()
    {
        OrderId = request.OrderId,
        PaymentMethodId = (int)EnumPaymentMethod.MoMo,
        TransId = "0",
        OrderInfo = orderInfo,
        Amount = Decimal.Parse(request.Amount),
        ExtraData = extractData,
        IsSuccess = false,
        CreatedUser = accountId
    };
    await _unitOfWork.OrderPaymentTransactions.AddAsync(orderPaymentTransaction);
    //Handle Payment Request
    var paymentRequest = new CreateGetwayRequestModel()
    {
        RequestId = orderPaymentTransaction.Id.ToString(),
        Amount = request.Amount,
        OrderId = orderPaymentTransaction.Id.ToString(),
        OrderInfo = orderInfo,
        RedirectUrl =  request.RedirectUrl , // redirect to weblink or applink after payment
        IpnUrl = ipnUrl,
        PartnerClientId = email,
        ExtraData = extractData,
        Language = request.Language
    };
    return paymentRequest;
}
```

Figure 48. Create Order with MoMo service system

The above code purpose is to save order information to the database and create a model to prepare and open a connection to the MoMo service.

About connection to MoMo system:

```
try
{
    LogInformation("MOMO_REQUEST", endpoint, postJsonString);
    HttpWebRequest httpWReq = (HttpWebRequest)WebRequest.Create(endpoint);
    var postData = postJsonString;
    var data = Encoding.UTF8.GetBytes(postData);
    httpWReq.ProtocolVersion = HttpVersion.Version11;
    httpWReq.Method = "POST";
    httpWReq.ContentType = "application/json";
    httpWReq.ContentLength = data.Length;
    httpWReq.ReadWriteTimeout = 30000;
    httpWReq.Timeout = 15000;
    Stream stream = httpWReq.GetRequestStream();
    stream.Write(data, 0, data.Length);
    stream.Close();
    HttpWebResponse httpWebResponse = (HttpWebResponse)httpWReq.GetResponse();
    string jsonresponse = string.Empty;
    using (var reader = new StreamReader(httpWebResponse.GetResponseStream()))
    {
        string temp = null;
        while ((temp = reader.ReadLine()) != null)
        {
            jsonresponse += temp;
        }
    }
    response.Success = true;
    response.Data = jsonresponse;
    return response;
}
```

Figure 49. Request to MoMo service example.

Description of each step in the code:

- Create HTTP request: Creates an HttpWebRequest object with the given endpoint.
- Set request properties: Method, Content, etc.
- Send request data: Open a stream to write data and sends the JSON data through this stream.
- Receive HTTP response: Receive response from MoMo service and open/ save it to json response.

3.2.2.6 Front-End: React TypeScript

React is a popular JavaScript library for building user interfaces, particularly single-page applications.

TypeScript, a statically typed superset of JavaScript, adds type safety and improves code maintainability.

Key Libraries and Tools

- **React Router**: For handling navigation and routing within the application.
- **Redux**: For state management.
- **Axios**: For making HTTP requests to the back-end API.
- **Antd Design**: For implementing responsive and aesthetically pleasing UI components.

Project Structure

- **src**: This is the main source folder where all the application code resides. It is further divided into subfolders:
    - **app**: This folder is likely to contain core application logic. It includes:
        - **api**: Contains files related to API calls, such as functions to interact with backend services.
        - **routes**: Manages the routing configuration for the application, defining the different routes and corresponding components.
        - **store**: Contains state management logic, possibly using a library like Redux or Context API.
    - **assets**: This folder holds static assets like images, fonts, or any other media that needs to be bundled with the application.
    - **components**: Contains reusable UI components used throughout the application. Each component is usually a React component or a collection of related components.
    - **containers**: Contains higher-level components that connect to the Redux store or manage state and logic for specific parts of the

application. Subfolders like filter, fnb-page-title, and fnb-text-input likely represent specific containers with their own logic and state management.

- **layouts**: Defines the layout components that structure the application's pages, including headers, footers, and sidebars.
- **contants** (likely a typo, should be **constants**): Holds constant values and configurations used across the application, such as URLs, keys, and static strings.
- **features**: Contains feature-specific logic and components, which might be encapsulated functionality or modules within the application.
- **hooks**: Contains custom React hooks that encapsulate reusable logic.
- **locales**: Manages localization files for internationalization, with subfolders for different languages (e.g., en for English, vi for Vietnamese).
- **pages**: Contains top-level components that represent different pages of the application. Each page usually corresponds to a route.
- **stylesheets**: This folder holds global stylesheet files, such as SCSS or CSS files, that apply styles across the application.
- **types**: Contains TypeScript type definitions and interfaces to ensure type safety and autocompletion within the codebase.
- **utils**: Contains utility functions and helpers used throughout the application. These functions perform common tasks and are shared among different parts of the application.
- **helpers.ts**: A file for helper functions that provide reusable logic for various parts of the application.

- **env.ts**: Contains environment-specific variables and configurations, typically used for setting up different environments like development, staging, and production.

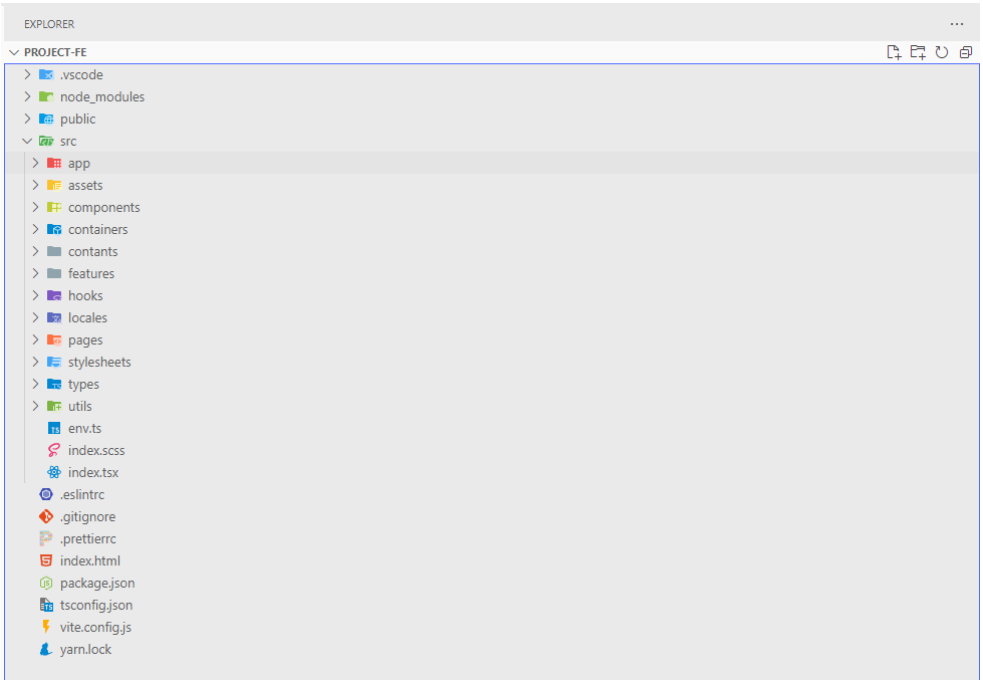


Figure 50. Front end project structure

I use Yarn for my project because yarn is used as a package manager for JavaScript projects because it offers several advantages over npm (Node Package Manager), including:

- **Speed**: Yarn is known for its fast performance, thanks to its parallel installation process and caching mechanisms. It can install packages more quickly than npm by caching downloaded packages and reusing them without having to download them again.
- **Deterministic Dependency Resolution**: Yarn uses a lockfile (yarn.lock) to ensure that the same dependencies are installed across different environments. This ensures that every developer on the project gets the

exact same version of dependencies, reducing the chances of "works on my machine" issues.

- **Offline Mode**: Since Yarn caches every package it downloads, it can install packages without an internet connection if the packages are already cached. This is particularly useful for developers working in environments with limited or unreliable internet access.
- **Enhanced Security**: Yarn checks the integrity of every installed package using checksums, ensuring that the package code hasn't been altered. This adds an extra layer of security to the dependency management process.
- **Better Dependency Management**: Yarn offers a more reliable and consistent way of managing dependencies with features like selective version resolutions, which allow developers to resolve conflicts by specifying the version of a dependency to use.
- **Workspaces**: Yarn supports monorepos through workspaces, allowing multiple projects to live together in the same repository and share dependencies more efficiently.
- **Active Community and Development**: Yarn has a strong community and is actively maintained, ensuring that it continues to evolve and improve over time.

3.2.2.7 Back-End: .NET Core API

.NET Core is a cross-platform, high-performance framework for building modern, cloud-based, and internet-connected applications.

Key Features

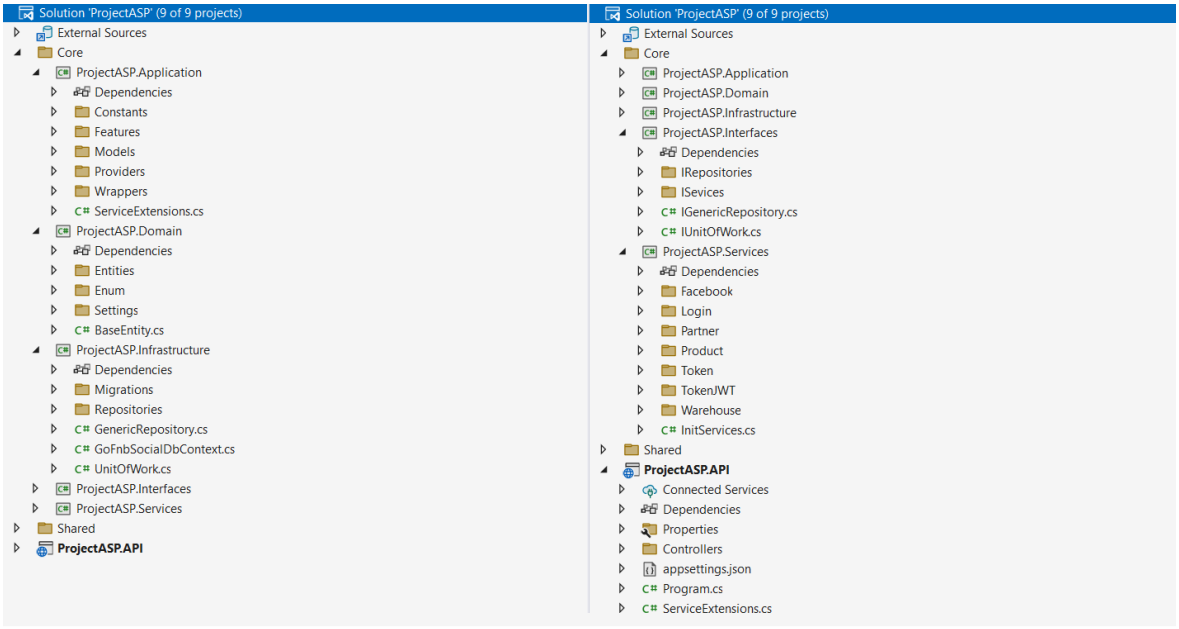


Figure 51. The folder structure and organization of the components in source code

Characteristics of Clean Architecture:

- **Separation of Layers:**
  - **Core (Domain Layer)**: Includes entities, enums, and core domain logic. This is the center of the application and does not depend on anything else.
  - **Application Layer**: Contains use cases, services, models, and business logic specific to the application. It can call the domain layer but does not depend on any infrastructure.
  - **Infrastructure Layer**: Contains data-related components like repositories, database contexts, and services that interact with the database. It depends on the domain and application layers.
  - **API Layer**: Includes controllers, API endpoints, and specific configurations for interacting with users or other systems.
- **Framework Independence:**
  - Frameworks are only used in the outer layers (e.g., the API layer) and do not affect the core logic of the application.

  - This allows for easy changes or updates to the framework without impacting other parts of the system.
- **Dependency Inversion Principle:**
  - Dependencies are defined in higher layers (such as interfaces in the domain layer), while lower layers (like infrastructure) implement these interfaces.
  - This ensures that core layers do not depend on the specifics of lower layers.

3.2.2.8 Back-End: .NET Core API

Specifically for my project:

- **Core Layer**:
  - **ProjectASP.Domain**: Contains entities, enums, and core logic of the application.
    - **Dependencies**: Manages dependencies specific to the domain layer.
    - **Entities**: Contains domain entities representing the core business objects.
    - **Enum**: Contains enumeration types used across the domain.
    - **Settings**: Contains configuration settings specific to the domain.
    - **BaseEntity.cs**: A base class for domain entities, typically including common properties like ID, created date, and modified date.

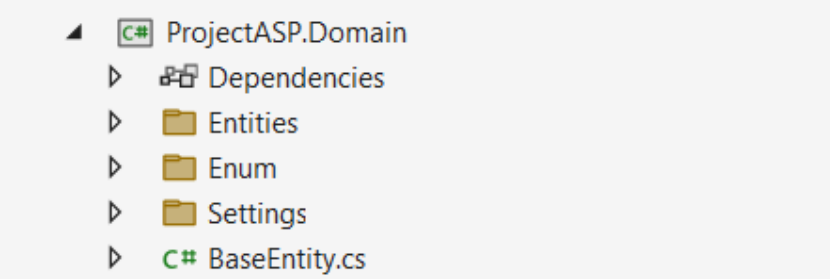


Figure 52. Domain layer

- **ProjectASP.Application**: Contains use cases, services, and business logic specific to the application.
  - **Dependencies**: Contains classes and interfaces for managing dependencies and services used in the application layer.
  - **Constants**: Holds constant values and static configurations used throughout the application.
  - **Features**: Contains the business logic and use cases. Each feature might represent a specific functionality of the application.
  - **Models**: Contains the data models representing the structure of data used within the application.
  - **Providers**: Contains classes that provide specific services or data, such as external API integrations or data providers.
  - **Wrappers**: Contains classes that wrap around external services or libraries to provide a unified interface for the rest of the application.
  - **ServiceExtensions**.cs: Contains extension methods for setting up and configuring services within the application, typically used in the Startup class for dependency injection.

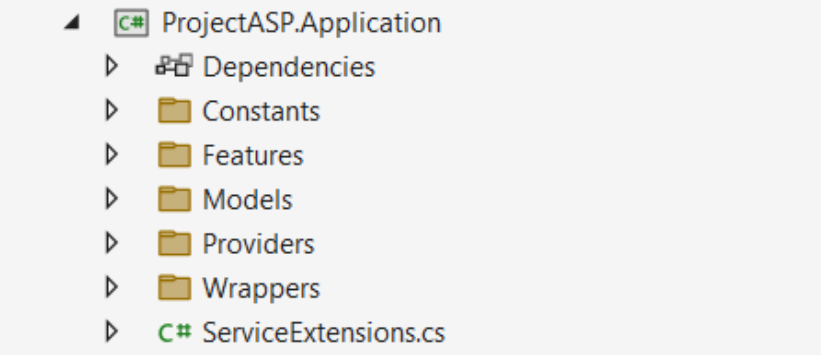


Figure 53. Application layer

- **Infrastructure Layer:**
  - **ProjectASP.Infrastructure**: Contains repositories, database contexts, and data access-related components.
    - **Dependencies**: Contains infrastructure-related dependencies and services.
    - **Migrations**: Manages database migration scripts and tools for schema changes.
    - **Repositories**: Contains repository classes that handle data access logic.
    - **GenericRepository.cs**: A generic implementation providing common data access methods for different entities.
    - **GoFnbSocialDbContext.cs**: The Entity Framework DbContext class representing the database session and providing access to entities.
    - **UnitOfWork.cs**: Implements the Unit of Work pattern to coordinate the work of multiple repositories sharing a single database context.

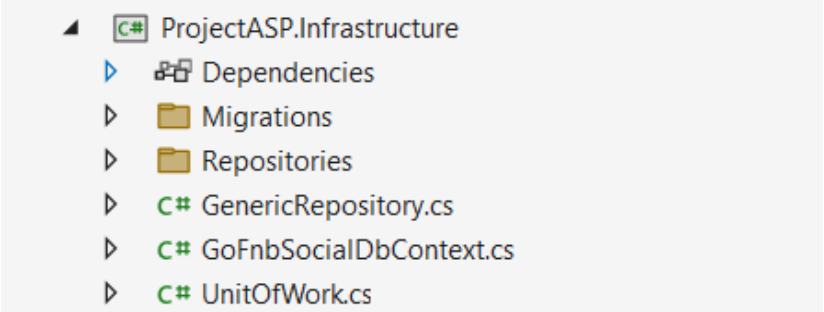


Figure 54. Infrastructure layer

- **Interface Layer:**
  - **ProjectASP.Interfaces**: Contains interfaces that define contracts for repositories and services, ensuring abstraction and independence from specific implementations.
    - **IRepositories**: Manages interface of repository.
    - **IServices**: Manages interface of service.

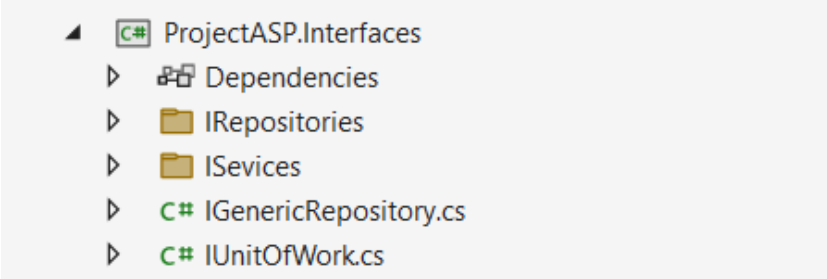


Figure 55. Interfaces layer

- **Service Layer:**
  - **ProjectASP.Services**: Contains specific services for different functionalities of the application (e.g., Login, Product, …).
    - Dependencies: Manages dependencies specific to the service layer.
    - Login: Contains services related to user authentication and login.
    - Product: Contains services related to product management.

- TokenJWT: Contains services specific to JWT (JSON Web Token) handling.
- InitServices.cs: Likely contains initialization logic for services, setting up configurations and dependencies.

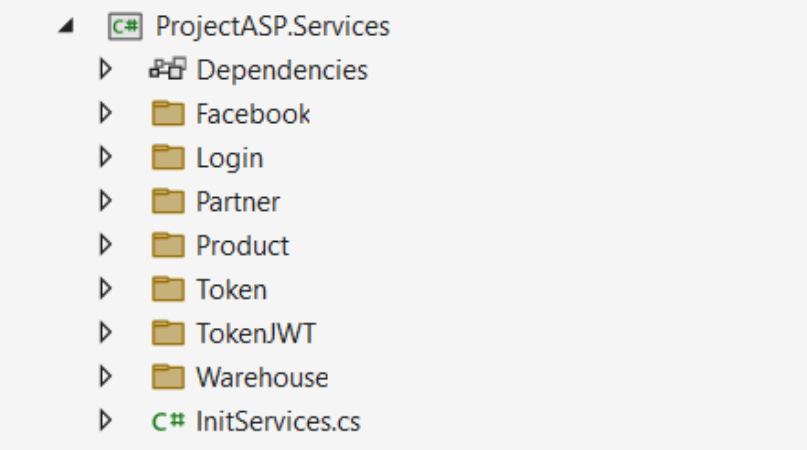


Figure 56. Services layer.

- **API Layer:**
  - **ProjectASP.API**: Contains controllers, API endpoints, and configuration for the API application, interacting with users or other systems.
    - **Connected Services**: Manages connected services such as third-party APIs, external data sources, or other service dependencies.
    - **Dependencies**: Contains dependency management and service configuration for the API project.
    - **Properties**: Contains project properties and settings.
    - **Controllers**: Contains API controller classes defining the endpoints for the web API, handling HTTP requests and responses.
    - **appsettings.json**: Configuration file for the API project, typically containing settings for database connections,

application configurations, and other environment-specific settings.

- **Program.cs**: The main entry point for the API application, configuring and running the web host.
- **ServiceExtensions.cs**: Contains extension methods for setting up and configuring services specific to the API project, typically used in the Program or Startup class for dependency injection.

Figure 57. API layer.

- **Shared Layer:**
  - **ProjectASP.Common, ProjectASP.Loging**: Contains utilities, helpers, and shared components used across different layers of the application.

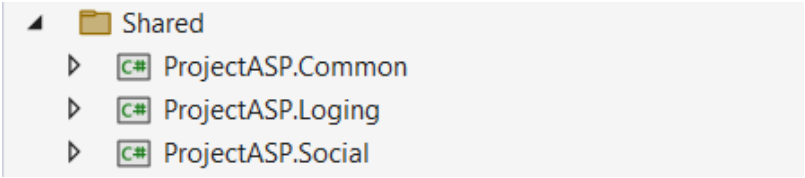


Figure 58. Shared layer

Security

Security is a critical aspect of the project, ensuring that sensitive information is protected.

Authentication and Authorization

- **JWT (JSON Web Token)**: Used for secure authentication.

- Login: need some information for Generate token.

```
public class TokenJWTService : ITokenJWTService
    {
        private const string SecretKey = "HuyNguyenQuang-Panda-hL3kD5b8sDf7eR5kG8uJ2wQ1yZ6sX3aL9dE7gF6kH1vN4uT7mP4sQ1tX9rG3bS5a";
        public Task<string> GenerateToken(Guid? id, Guid? storeId, string username, string password, string fullname)
        {
            var tokenHandler = new JwtSecurityTokenHandler();
            var key = Encoding.ASCII.GetBytes(SecretKey);
            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(new[]
                {
                new Claim("Id", id.ToString()),
                new Claim("StoreId", storeId.ToString()),
                new Claim("Username", username),
                new Claim("Password", password),
                new Claim("Fullname", fullname)
            }),
                Expires = DateTime.UtcNow.AddHours(1),
                SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
            };
            var token = tokenHandler.CreateToken(tokenDescriptor);
            var tokenString = tokenHandler.WriteToken(token);
            return Task.FromResult(tokenString);
        }
    }
```

Figure 59. Use JWT for web token login authentication.

- **Password Hashing:**
    - Passwords are hashed using a secure algorithm (e.g., bcrypt) before storing them in the database.

```
var hash = (new PasswordHasher<Account>()).HashPassword(null, request.Password);

Account registerAccountModel = new()
{
    StoreId = store.Id,
    Username = request.Email,
    Password = hash,
    PhoneNumber = request.PhoneNumber == null ? "" : request.PhoneNumber,
    Thumbnail = "https://www.vietnamworks.com/hrinsider/wp-content/uploads/2023/12/anh-sieu-cute-001-1.jpg",
    ValidateCode = "",
    Gender = Domain.Enums.EnumGender.Other,
    FullName = request.FullName
};

await _unitOfWork.Accounts.AddAsync(registerAccountModel);
```

Figure 60. Hashing password before save it to database

3.2.2.9 Database - SQL Server

SQL Server is a robust and scalable relational database management system.

Database Schema Design – POS Web System

- The database schema includes the following tables:

- **Account**: An account maintains user details and credentials, typically including the following fields:
  - **Id**: A unique identifier for the account.
  - **StoreId**: A reference to the store linked to the account.
  - **Code**: A distinct code for the account.
  - **Username**: The user's login identifier.
  - **Password**: The user's secret key.
  - **FullName**: The user's complete name.
  - **ValidateCode**: A code used for verification purposes.
  - **EmailConfirmed**: Indicates if the email address has been verified.
  - **PlatformId**: An identifier for the platform.
  - **IsActivated**: Shows whether the account is active.
  - **PhoneNumber**: The user's contact number.
  - **Thumbnail**: A URL or path to the user's profile image.
  - **Birthday**: The user's date of birth.
  - **Gender**: The user's sex.
  - **LastSavedUser**: The last person who updated the record.
  - **LastSavedTime**: The last time the record was modified.
  - **CreatedUser**: The person who created the record.
  - **CreatedTime**: The timestamp when the record was created.
  - **IsDeleted**: A flag indicating if the record is marked as deleted.

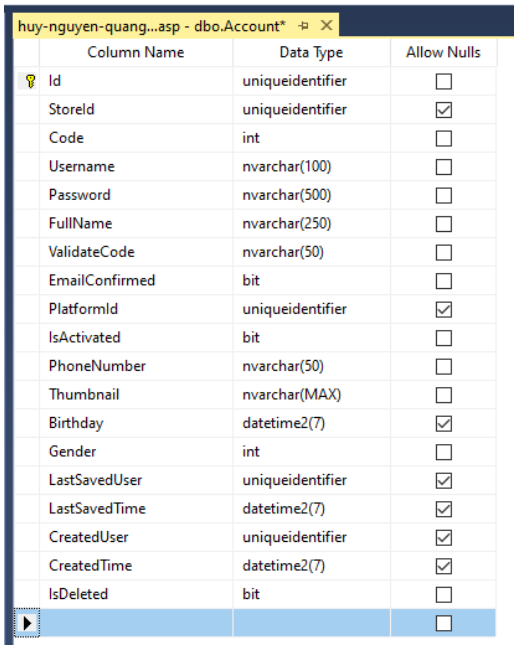


Figure 61. Account table

- Store: This table contains details about store locations, including logos and additional information. Fields include:
  - **Id**: A unique identifier for the store.
  - **AddressId**: A reference to the store's address.
  - **Code**: A distinct code for the store.
  - **Logo**: A URL or path to the store's logo.
  - **Title**: The title or name of the store.
  - **IsActivated**: Indicates whether the store is active.
  - **LastSavedUser**: The last person who updated the record.
  - **LastSavedTime**: The most recent time the record was updated.
  - **CreatedUser**: The person who created the record.
  - **CreatedTime**: The timestamp when the record was created.
  - **IsDeleted**: A flag indicating if the record is marked as

deleted**.**

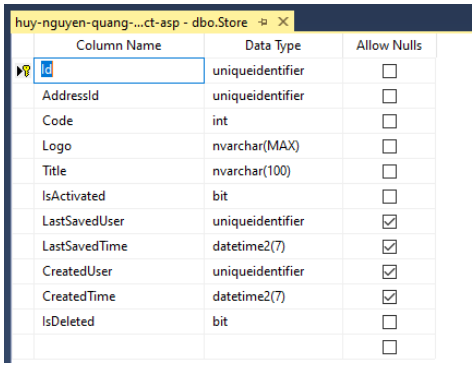


Figure 62. Store table

- **Product**: This table contains information about products. **Product** generally includes:
  - **Id**: A unique identifier for the product.
  - **StoreId**: A reference to the store the product belongs to.
  - **UnitId**: A reference to the unit of measure for the product.
  - **TaxId**: A reference to the tax information.
  - **StatusId**: The status of the product (e.g., active, inactive).
  - **Name**: The name of the product.
  - **Description**: A description of the product.
  - **Thumbnail**: A URL or path to the product's thumbnail image.
  - **IsTopping**: Indicates whether the product is a topping.
  - **IsActive**: Indicates whether the product is active.
  - **IsPrintStamp**: Indicates whether the product has a print stamp.
  - **Code**: A unique code for the product.
  - **Abbreviation**: An abbreviation for the product.

- **LastSavedUser**: The last person who updated the record.
- **LastSavedTime**: The most recent time the record was updated.
- **CreatedUser**: The person who created the record.
- **CreatedTime**: The timestamp when the record was created.
- **IsDeleted**: A flag indicating if the record is marked as deleted.

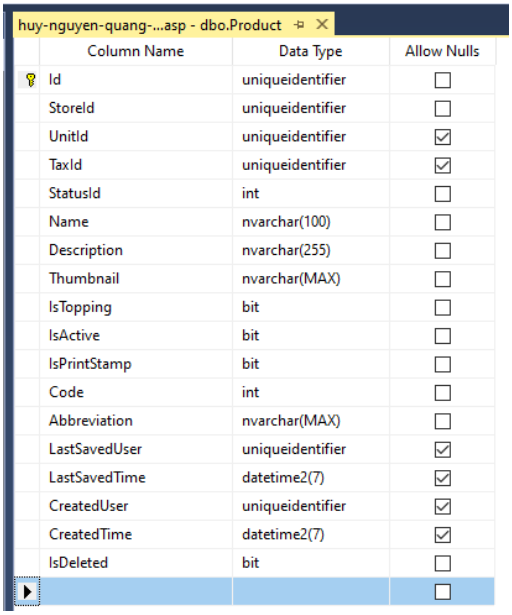


Figure 63. Product table

- **Order**: This table stores order details and transaction history. Typical fields include:
  - **Id**: Unique identifier for the order.
  - **StoreId**: Reference to the store where the order was placed.
  - **Note**: Additional notes for the order.
  - **Code**: Unique code for the order.
  - **OriginalPrice**: Original price of the order before discounts

or adjustments.

- **CashierName**: Name of the cashier who handled the order.
- **ReceivedAmount**: Amount received for the order.
- **Change**: Change given back to the customer.
- **TotalAmount**: Total amount of the order.
- **Status**: Status of the order (e.g., pending, completed).
- **LastSavedUser**: The last user who updated the record.
- **LastSavedTime**: The last time the record was updated.
- **CreatedUser**: The user who created the record.
- **CreatedTime**: The time when the record was created.
- **IsDeleted**: Soft delete flag.

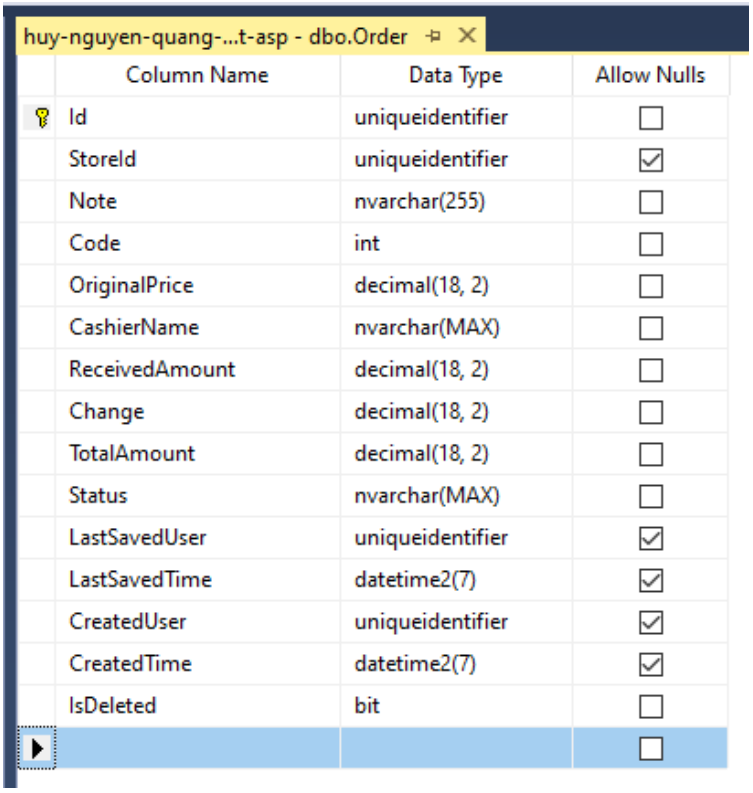
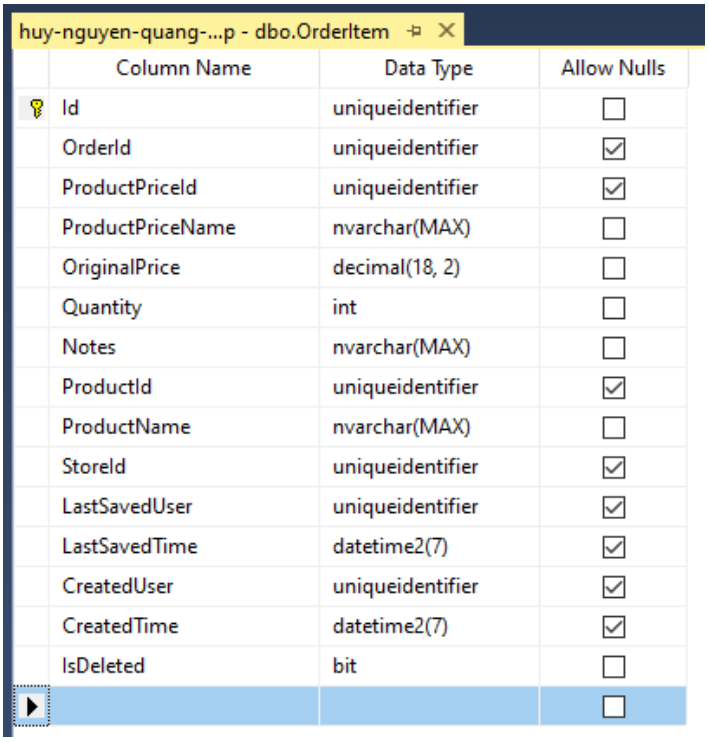


Figure 64. Order table

- **Order Item:** This table stores information about individual items within an order. It typically includes:
  - **Id**: Unique identifier for the order item.
  - **OrderId**: Reference to the order this item belongs to.
  - **ProductPriceId**: Reference to the product price.
  - **ProductPriceName**: Name of the product price.
  - **OriginalPrice**: Original price of the item.
  - **Quantity**: Quantity of the item ordered.
  - **Notes**: Additional notes for the order item.
  - **ProductId**: Reference to the product.
  - **ProductName**: Name of the product.
  - **StoreId**: Reference to the store where the order was placed.
  - **LastSavedUser**: The last user who updated the record.
  - **LastSavedTime**: The last time the record was updated.
  - **CreatedUser**: The user who created the record.
  - **CreatedTime**: The time when the record was created.
  - **IsDeleted**: Soft delete flag.


huy-nguyen-quang-...p - dbo.OrderItem

Column Name	Data Type	Allow Nulls
Id	uniqueidentifier	☐
OrderId	uniqueidentifier	☑
ProductPriceId	uniqueidentifier	☑
ProductPriceName	nvarchar(MAX)	☐
OriginalPrice	decimal(18, 2)	☐
Quantity	int	☐
Notes	nvarchar(MAX)	☐
ProductId	uniqueidentifier	☑
ProductName	nvarchar(MAX)	☐
StoreId	uniqueidentifier	☑
LastSavedUser	uniqueidentifier	☑
LastSavedTime	datetime2(7)	☑
CreatedUser	uniqueidentifier	☑
CreatedTime	datetime2(7)	☑
IsDeleted	bit	☐
		☐

Figure 65. Order Item table

Data Access Layer

- The data access layer utilizes Entity Framework Core, which offers a collection of APIs for querying and managing data.

Database Schema Design – Admin Web System

- The database schema includes the following tables:
    - **Account**: Stores user information and credentials. It typically includes fields such as:
        - **Id**: Unique identifier for the account.
        - **StoreId**: Reference to the store the account is associated with.
        - **Code**: Unique code for the account.
        - **Username**: User's login name.

- **Password**: User's password.
- **FullName**: User's full name.
- **ValidateCode**: Code used for validation purposes.
- **EmailConfirmed**: Indicates if the email is confirmed.
- **PlatformId**: Identifier for the platform.
- **IsActivated**: Indicates if the account is activated.
- **PhoneNumber**: User's phone number.
- **Thumbnail**: URL or path to the user's thumbnail image.
- **Birthday**: User's birth date.
- **Gender**: User's gender.
- **LastSavedUser**: The last user who updated the record.
- **LastSavedTime**: The last time the record was updated.
- **CreatedUser**: The user who created the record.
- **CreatedTime**: The time when the record was created.
- **IsDeleted**: Soft delete flag.

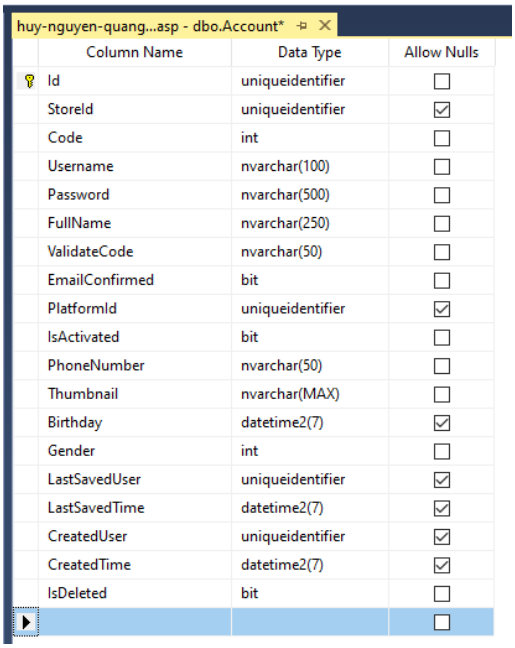


Figure 66. Account table

- **Store**: This table stores details about different store locations,

including logos and additional information. Fields might include:

- **Id**: Unique identifier for the store.
- **AddressId**: Reference to the store's address.
- **Code**: Unique code for the store.
- **Logo**: URL or path to the store's logo.
- **Title**: Title or name of the store.
- **IsActivated**: Indicates if the store is activated.
- **LastSavedUser**: The last user who updated the record.
- **LastSavedTime**: The last time the record was updated.
- **CreatedUser**: The user who created the record.
- **CreatedTime**: The time when the record was created.
- **IsDeleted**: Soft delete flag.

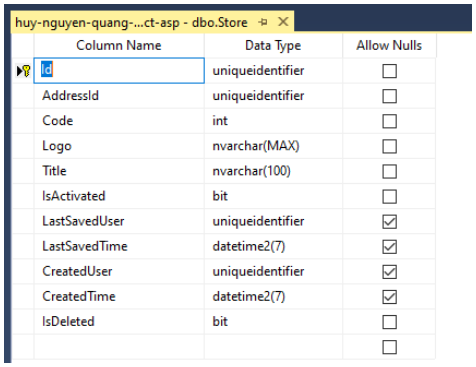


Figure 67. Store table

- **Product**: This table stores product information. It generally includes: Id: Unique identifier for the product.
  - **StoreId**: Reference to the store the product belongs to.
  - **UnitId**: Reference to the unit of measure for the product.
  - **TaxId**: Reference to the tax information.

- **StatusId**: Status of the product (e.g., active, inactive).
- **Name**: Name of the product.
- **Description**: Description of the product.
- **Thumbnail**: URL or path to the product's thumbnail image.
- **IsTopping**: Indicates if the product is a topping.
- **IsActive**: Indicates if the product is active.
- **IsPrintStamp**: Indicates if the product has a print stamp.
- **Code**: Unique code for the product.
- **Abbreviation**: Abbreviation for the product.
- **LastSavedUser**: The last user who updated the record.
- **LastSavedTime**: The last time the record was updated.
- **CreatedUser**: The user who created the record.
- **CreatedTime**: The time when the record was created.
- **IsDeleted**: Soft delete flag.

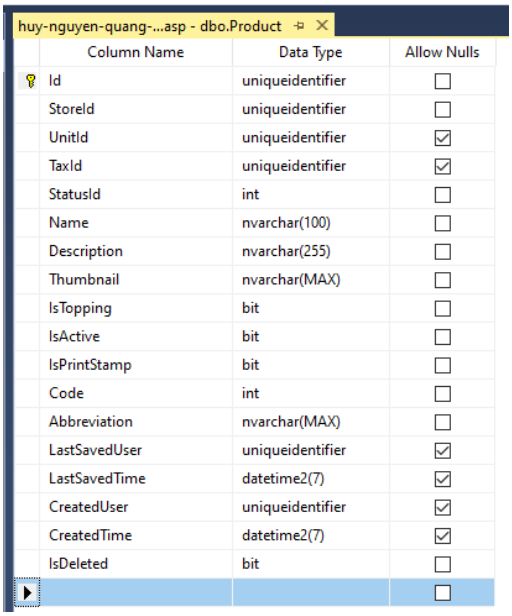


Figure 68. Product table

- **Material**: This table stores raw material information. Fields might include:
  - **Id**: Unique identifier for the material.
  - **StoreId**: Reference to the store the material belongs to.
  - **UnitId**: Reference to the unit of measure for the material.
  - **Code**: Unique code for the material.
  - **Name**: Name of the material.
  - **Description**: Description of the material.
  - **Sku**: Stock Keeping Unit identifier.
  - **MinQuantity**: Minimum quantity for the material.
  - **Quantity**: Available quantity of the material.
  - **CostPerUnit**: Cost per unit of the material.
  - **IsActive**: Indicates if the material is active.
  - **HasExpiryDate**: Indicates if the material has an expiry date.
  - **Thumbnail**: URL or path to the material's thumbnail image.
  - **LastSavedUser**: The last user who updated the record.
  - **LastSavedTime**: The last time the record was updated.
  - **CreatedUser**: The user who created the record.
  - **CreatedTime**: The time when the record was created.
  - **IsDeleted**: Soft delete flag.

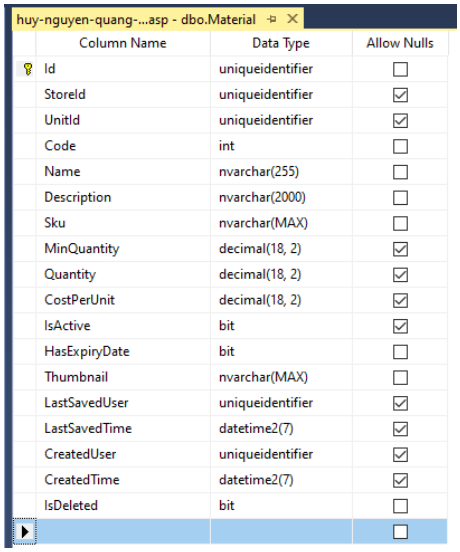


Figure 69. Material table

- **Unit:** This table stores unit information for products or materials. Fields might include:
  - **Id**: Unique identifier for the unit.
  - **StoreId**: Reference to the store the unit belongs to.
  - **Code**: Unique code for the unit.
  - **Name**: Name of the unit.
  - **LastSavedUser**: The last user who updated the record.
  - **LastSavedTime**: The last time the record was updated.
  - **CreatedUser**: The user who created the record.
  - **CreatedTime**: The time when the record was created.
  - **IsDeleted**: Soft delete flag.

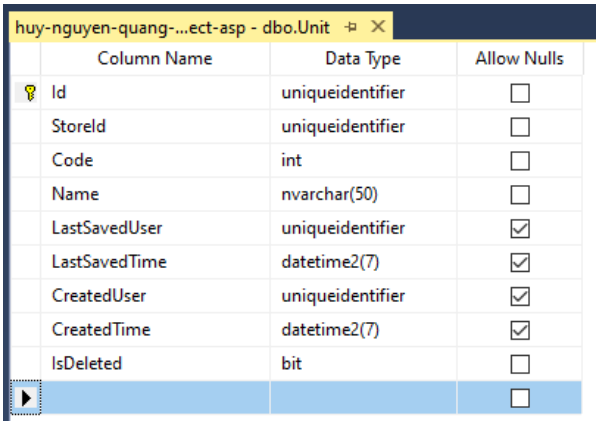


Figure 70. Unit table

- **Product Price**: This table stores pricing information for products. It typically includes:
  - **Id**: Unique identifier for the product price.
  - **ProductId**: Reference to the product.
  - **Code**: Unique code for the product price.
  - **PriceName**: Name of the price.
  - **PriceValue**: Value of the price.
  - **StoreId**: Reference to the store where the price is applicable.
  - **LastSavedUser**: The last user who updated the record.
  - **LastSavedTime**: The last time the record was updated.
  - **CreatedUser**: The user who created the record.
  - **CreatedTime**: The time when the record was created.
  - **IsDeleted**: Soft delete flag.

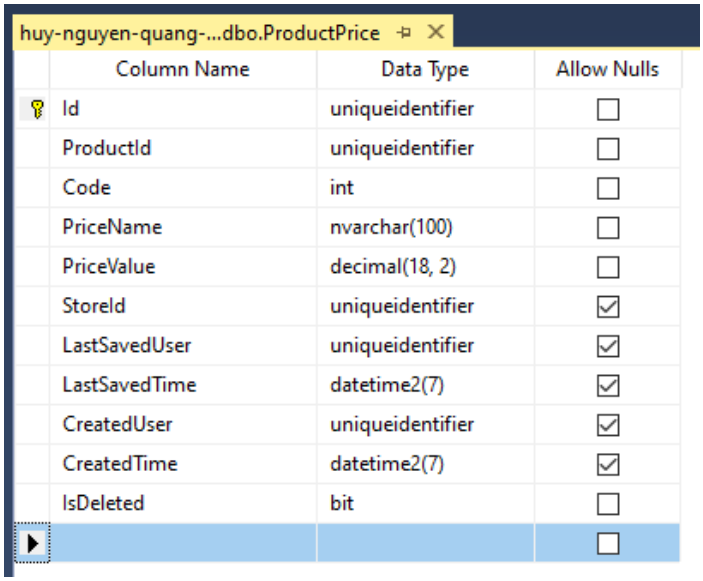


Figure 71. Product Price table.

- **Product Material Price:** This table stores pricing information for materials used in products. Fields might include:
  - **Id**: Unique identifier for the product material price.
  - **ProductPriceId**: Reference to the product price.
  - **MaterialId**: Reference to the material.
  - **Quantity**: Quantity of the material.
  - **StoreId**: Reference to the store where the price is applicable.
  - **UnitId**: Reference to the unit of measure.
  - **LastSavedUser**: The last user who updated the record.
  - **LastSavedTime**: The last time the record was updated.
  - **CreatedUser**: The user who created the record.
  - **CreatedTime**: The time when the record was created.
  - **IsDeleted**: Soft delete flag.

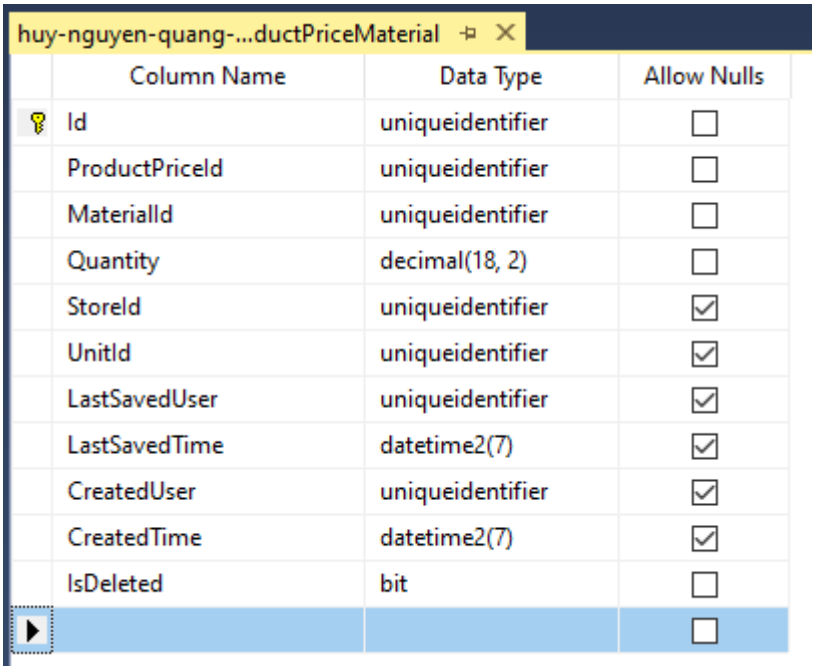


Figure 72. Product Material Price table.

Data Access Layer

- The data access layer is implemented using Entity Framework Core, which provides a set of APIs for querying and manipulating data.

3.3 Use Cases and Flow Charts

3.3.1 POS Web App

3.3.1.1 Use Case

Use Case	**POS Admin Registration**
Actor	POS Admin
Description	Allows a new pos admin to register an account on the system.
Precondition	The pos admin must have valid information to register
Postcondition	The pos admin account is created and saved in the system

Main Flow	1. POS Admin navigates to the registration page. 2. POS Admin fills in the registration form with their full name, store name, email, password, and confirms the password. 3. POS Admin submits the registration form. 4. System validates the inputs. 5. System creates a new pos admin account. 6. System confirms successful registration to the pos admin.

Use case 1. POS Register

Use Case	**POS Admin Login**
Actor	POS Admin
Description	Allows an existing pos admin to log in to the system
Precondition	POS Admin must have a registered account
Postcondition	POS Admin is logged into the system and can create order and view order
Main Flow	1. Admin navigates to the login page. 2. Admin enters their email and password. 3. Admin submits the login form. 4. System validates the credentials. 5. System logs in the admin and redirects to the Product List – Home Page

Use case 2. POS login

Use Case	**Create Order**
Actor	Cashier

	Admin
Description	Create an order is created and saved in the system
Precondition	The user is logged into the system The user has access to the menu and can view available items
Postcondition	An order is created and saved in the system
Main Flow	1. **User navigates to the Home Pagbe.** - The user accesses the pos website and goes to the order page. 2. **User views the menu.** - The system displays a list of available items (e.g., Cafe Đá, Cafe Đen) with their descriptions and prices. 3. **User selects items to add to the cart.** - The user clicks on "Add to Cart" for the desired items. - The system updates the cart with the selected items. 4. **User adjusts quantities in the cart.** - The user can increase or decrease the quantity of each item in the cart using the plus and minus buttons. - The system updates the total price based on the quantities selected. 5. **User reviews the order.** - The user checks the items and quantities in the cart to ensure they are correct. - The system displays the order summary, including item names, quantities, individual prices, and the total price. 6. **User creates the order.**

	- The user clicks the "Create Order" button. - The system processes the order and saves it in the database. 7. **System confirms the order.** - The system displays an order confirmation message to the user.

Use case 3. Create Orders

Use Case	**View Orders**
Actor	Cashier Admin
Description	The user can view the all orders
Precondition	The user is logged into the system The user has created one or more orders
Postcondition	The user can view the all orders
Main Flow	1. User navigates to the orders page. 2. System displays the list of orders.

Use case 4. View Orders.

3.3.1.2 Flow Charts

Create Account flow chart

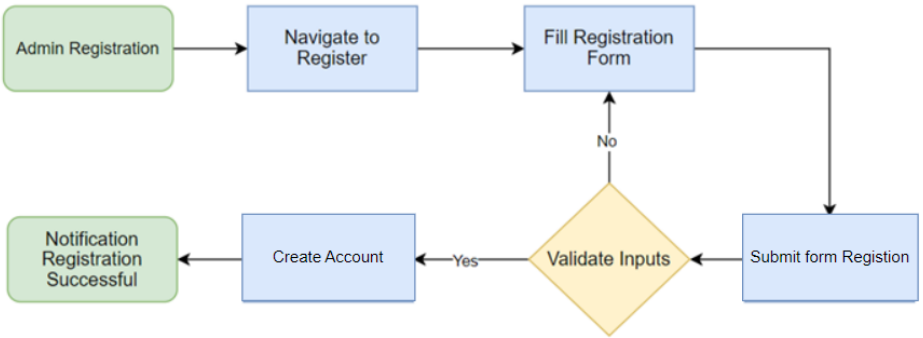


Figure 73. Create account POS flow chart

- **Step-by-Step Explanation**:

1. **Admin Registration**: This is the starting point where the admin begins the process of registration.
2. **Navigate to Register Page**: The admin navigates to the register page.
3. **Fill Registration Form**: Admin fill all information of form on a register page.
4. **Submit form Registration**: After admin fill all fields required that click on button Register.
5. **Validate Input:** Check all fields have data.
6. **Create Account:** Save Store information and Admin account to database.

Login POS flow chart

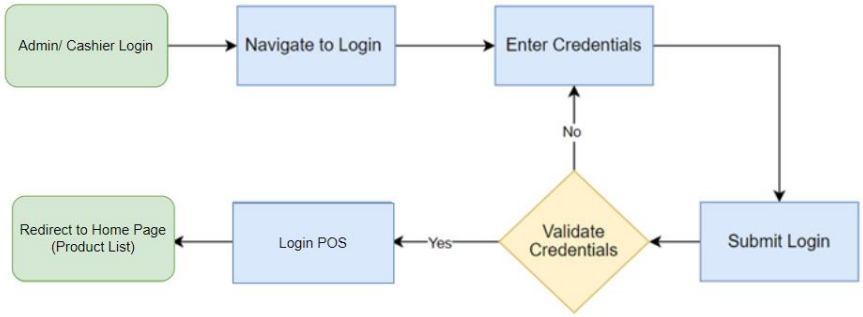


Figure 74. Login POS flow chart

- **Step-by-Step Explanation**:

1. **Admin/ Cashier Login:** This is the starting point where the admin begins the process of Login system.
2. **Navigate to Login Page**: The admin navigates to the login page.
3. **Enter Credentials:** The admin input their username and password.
4. **Submit Login:** The admin clicks on Login button.

5. **Validate Credentials:** Server checking username and password that correct or un-correct.
6. **Login POS:** When user name and password that user was input is matching with database. Server will send a token for front end to check credentials.

Create Order flow chart

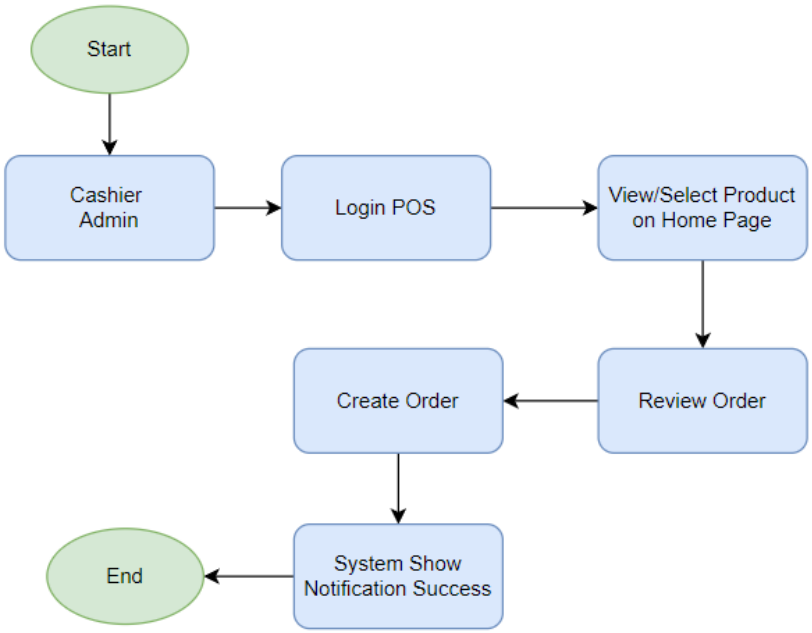


Figure 75. Create Order POS flow chart

- **Step-by-Step Explanation:**

1. **Admin/Cashier Login**: This is the starting point where the admin or cashier begins the process by logging into the system.
2. **Login POS:** When the username and password match the database, the server sends a token to the front end to verify the credentials and navigate to Home Page.
3. **View/Select Product on Home Page**: The admin or cashier views and selects a product on the home page.
4. **Review Order**: The admin or cashier reviews the selected order.

5. **Create Order**: The admin or cashier creates the order.
6. **System Show Notification Success**: The system shows a notification indicating the order was successfully created.

View Orders flow chart

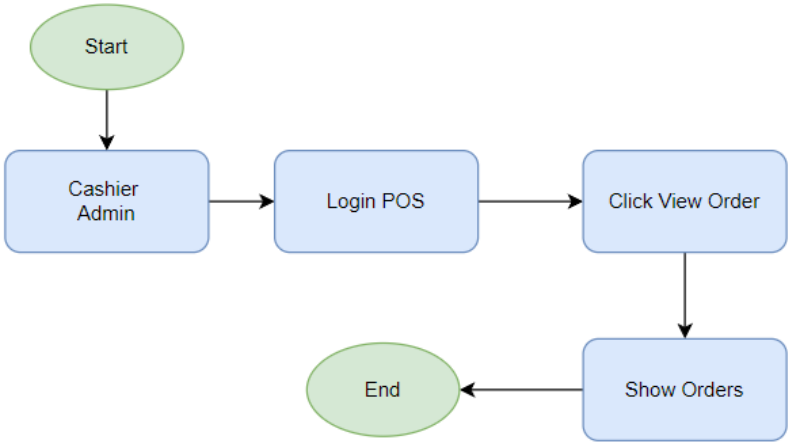


Figure 76. View Order flow chart

- **Step-by-Step Explanation:**
  1. **Admin/Cashier Login**: This is the starting point where the admin or cashier begins the process by logging into the system.
  2. **Login POS**: The admin or cashier logs into the POS system.
  3. **Click View Order**: The admin or cashier clicks on the "View Order" button.
  4. **Show Orders**: The system displays the list of orders.

3.3.2 Admin Web System

3.3.2.1 Use Case

Use Case	**Admin Registration**
Actor	Admin
Description	Allows a new admin to register an account on the system

Precondition	The admin must have valid details to register
Postcondition	The admin account is created and saved in the system
Main Flow	1. Admin navigates to the registration page. 2. Admin fills in the registration form with their full name, store name, email, password, and confirms the password. 3. Admin submits the registration form. 4. System validates the inputs. 5. System creates a new admin account. 6. System confirms successful registration to the admin.

Use case 5. Admin register

Use Case	**Admin Login**
Actor	Admin
Description	Allows an existing admin to log in to the system
Precondition	Admin must have a registered account
Postcondition	Admin is logged into the system and can manage materials and products
Main Flow	1. Admin navigates to the login page. 2. Admin enters their email and password. 3. Admin submits the login form. 4. System validates the credentials. 5. System logs in the admin and redirects to the dashboard.

Use case 6. Admin login

Use Case	**Manage Materials**
Actor	Admin
Description	Allows the admin to add, edit, and delete materials in the system
Precondition	Admin must be logged in
Postcondition	Materials are created/updated in the system
Main Flow	1. Admin navigates to the materials management page. 2. Admin chooses to add a material. 3. Admin fills in the details and submits the form. 4. System validates the inputs and adds the new entry to the database. 5. Admin chooses to edit an existing material. 6. Admin updates the details and submits the form. 7. System validates the inputs and updates the entry in the database. 8. Admin chooses to delete a material.

Use case 7. Manage materials

Use Case	**Manage Products**
Actor	Admin
Description	Allows the admin to add, edit, and delete products in the system
Precondition	Admin must be logged in
Postcondition	Products are created/updated in the system

Main Flow	1. Admin navigates to the products management page. 2. Admin chooses to add a product. 3. Admin fills in the details and submits the form. 4. System validates the inputs and adds the new entry to the database. 5. Admin chooses to edit an existing product. 6. Admin updates the details and submits the form. 7. System validates the inputs and updates the entry in the database. 8. Admin chooses to delete a product. 9. System removes the entry from the database after confirmation.

Use case 8. Manage product

3.3.2.2 Flow Charts

- **Admin Registration Flow Chart**

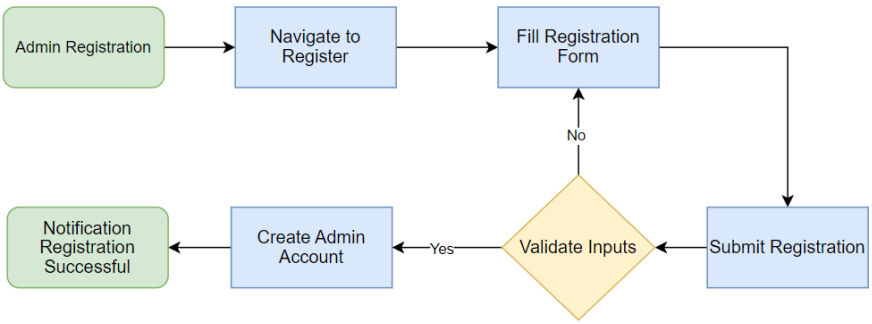


Figure 77. Admin Registration flow chart.

Step-by-Step Explanation:

1. **Admin Registration**: This is the starting point where an admin begins the registration process.
2. **Navigate to Register**: The admin navigates to the registration page.
3. **Fill Registration Form**: The admin fills out the registration form with the necessary details such as Full Name, Store Name, Email, Password, etc.
4. **Submit Registration**: After filling out the form, the admin submits the registration details.
5. **Validate Inputs**: The system checks the inputs provided by the admin. This includes verifying the format of the email, ensuring the password meets the security requirements, and checking for any missing fields.
    - **No**: If the inputs are not valid, the system prompts the admin to correct the errors and re-fill the registration form.
    - **Yes**: If the inputs are valid, the system proceeds to the next step.
6. **Create Admin Account**: Once the inputs are validated, the system creates a new admin account in the database.
7. **Notification Registration Successful**: After the account is successfully created, the system notifies the admin that the registration process was successful.

- **Admin Login Flow Chart**

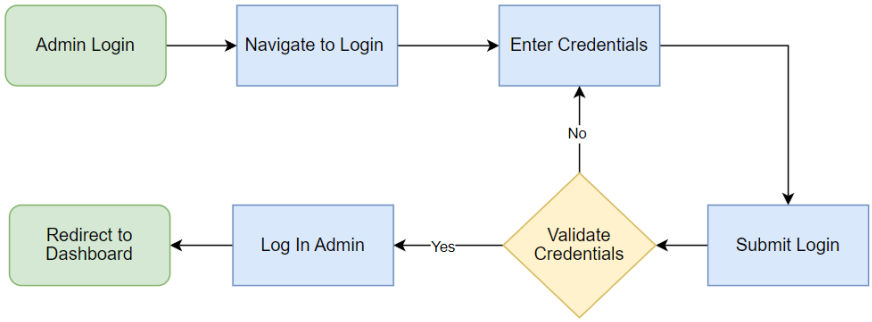


Figure 78. Admin Login flow chart.

Step-by-Step Explanation:

1. **Admin Login**: This is the starting point where an admin begins the login process.
2. **Navigate to Login**: The admin navigates to the login page of the system.
3. **Enter Credentials**: The admin enters their login credentials, typically a username and password.
4. **Submit Login**: After entering the credentials, the admin submits the login form.
5. **Validate Credentials**: The system checks the validity of the credentials provided by the admin.
    - **No**: If the credentials are not valid (e.g., incorrect username or password), the system prompts the admin to re-enter the correct credentials.
    - **Yes**: If the credentials are valid, the system proceeds to log in the admin.
6. **Log In Admin**: Once the credentials are validated, the system logs the admin into the system.
7. **Redirect to Dashboard**: After logging in, the system redirects the admin to the dashboard.

- **Admin Manage Material/Product flow chart**

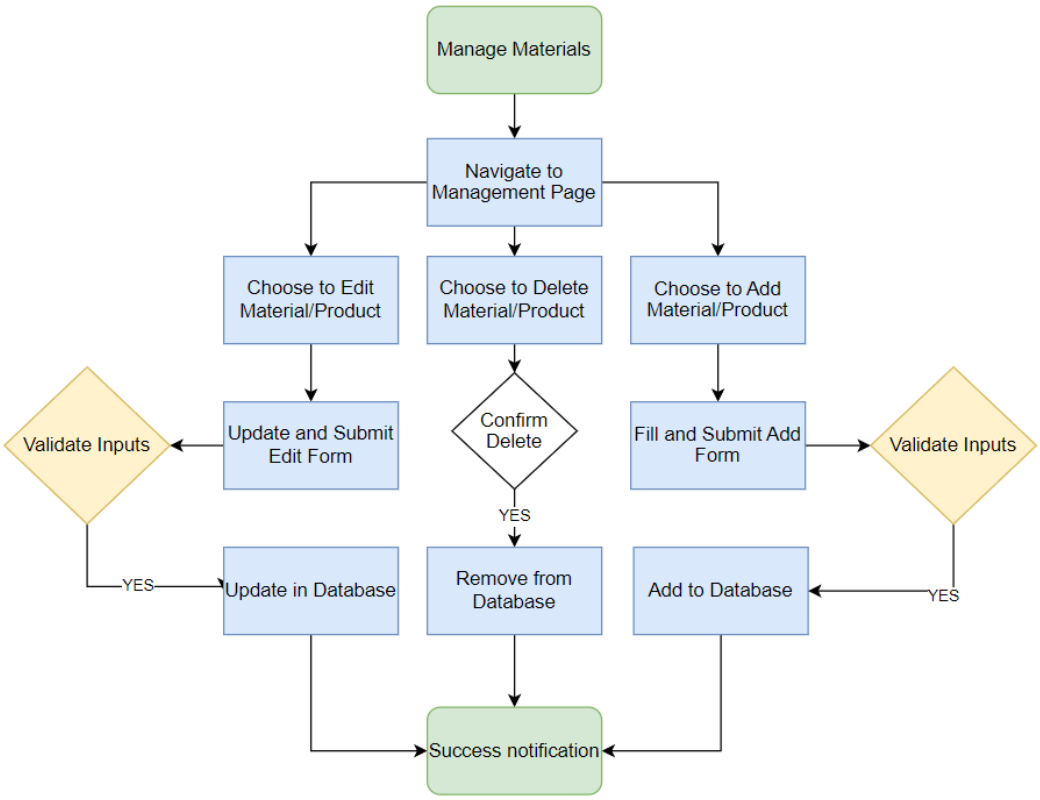


Figure 79. Manage material/ product flow chart

Step-by-Step Explanation:

1. **Manage Materials**: This is the starting point where the admin begins the process of managing materials or products.
2. **Navigate to Management Page**: The admin navigates to the materials or products management page.
3. **Choose to Edit Material/Product**: The admin selects an existing material or product to edit.
    - **Update and Submit Edit Form**: The admin updates the details of the selected material or product and submits the form.
    - **Validate Inputs**: The system validates the updated inputs.
        - **Yes**: If the inputs are valid, the system updates the entry in the database.

        - **Update in Database**: The system updates the material or product information in the database.
4. **Choose to Delete Material/Product**: The admin selects a material or product to delete.
    - **Confirm Delete**: The system prompts for confirmation to delete the selected material or product.
        - **Yes**: If confirmed, the system removes the entry from the database.
            - **Remove from Database**: The system deletes the material or product from the database.
5. **Choose to Add Material/Product**: The admin selects the option to add a new material or product.
    - **Fill and Submit Add Form**: The admin fills in the necessary details for the new material or product and submits the form.
    - **Validate Inputs**: The system validates the provided inputs.
        - **Yes**: If the inputs are valid, the system adds the new material or product to the database.
            - **Add to Database**: The system inserts the new material or product into the database.
6. **Success Notification**: After each successful operation (edit, delete, or add), the system sends a success notification to the admin.

3.4 Realize Projects And Demos

3.4.1 Admin Web System

3.4.1.1 Login Page

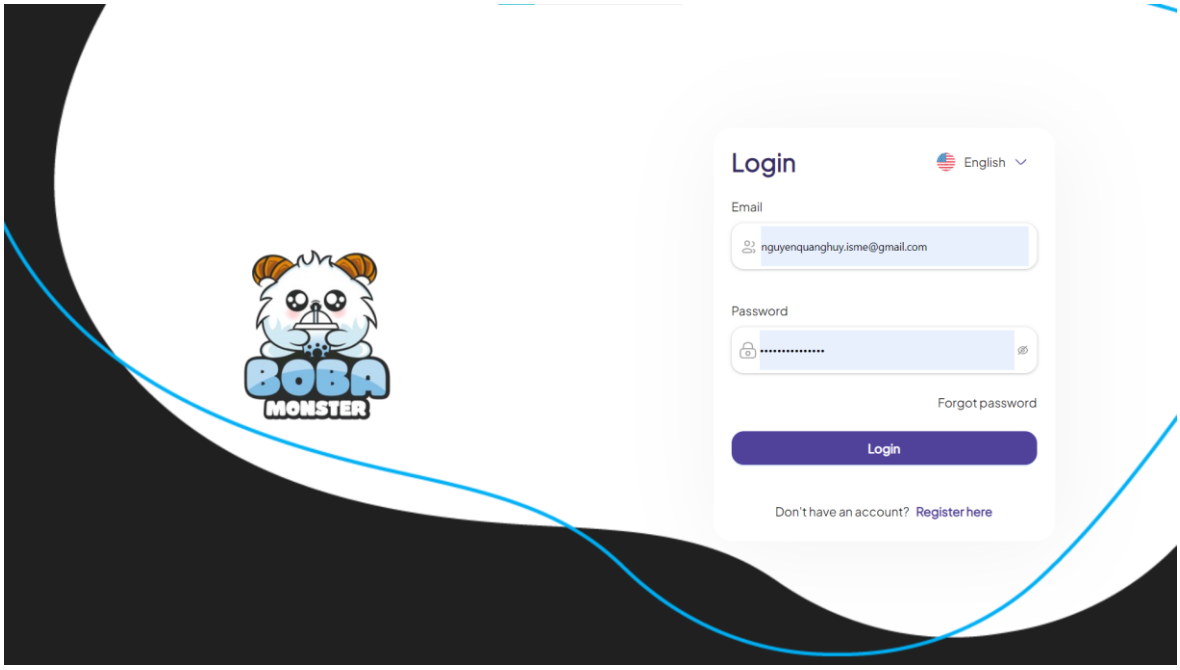


Figure 80. Admin login page

Admin login page description:

The admin login page features have a background with a login form situated in the center of the page, aligned to the right side.

This form comprises the following components:

- **Language Selector**: Located in the top right corner of the form, this option allows users to choose between **English** and **Vietnamese**. This feature switches languages based on locale files (en/vi).
- **Email Input Field**: A text box designed for users to input their email addresses.
- **Password Input Field**: This field is used for entering passwords required for logging in and includes an option to display or conceal the entered password.
- **Login Button**: After entering the email and password, users can click this button to authenticate and access their account.

- **Registration Link**: If the store owner or admin does not already have an account, this link can be used to sign up by providing the necessary store and personal information.

3.4.1.2 Admin Register Page

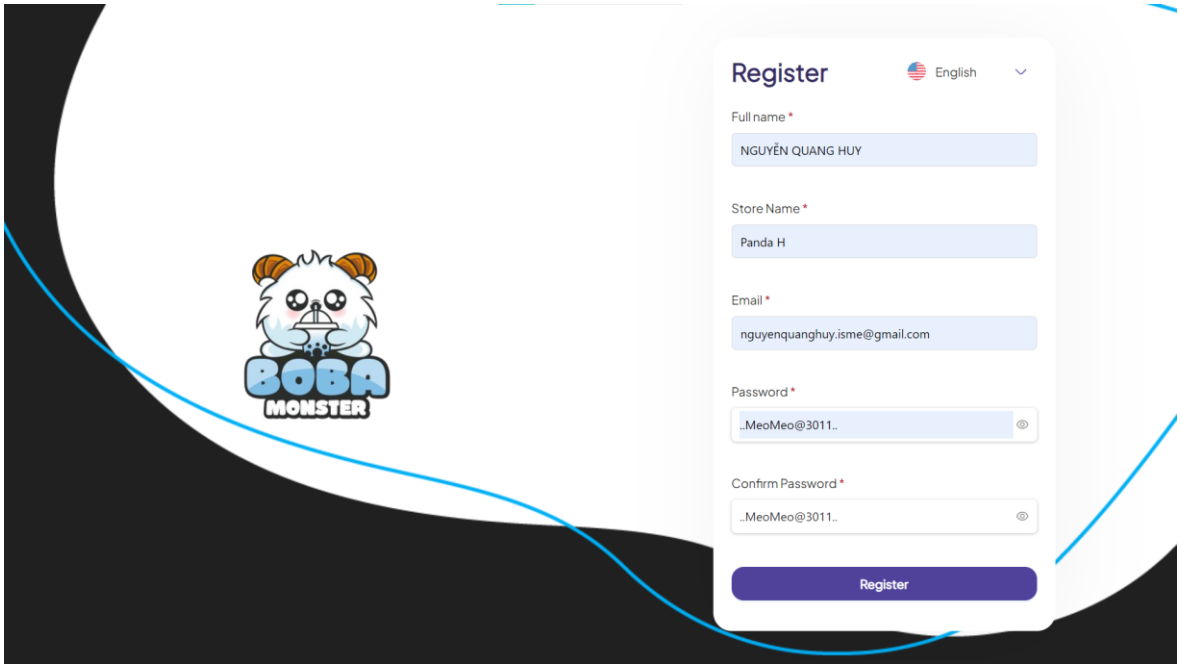


Figure 81. Admin register page.

Registration Page Description:

The registration page is centered around the registration form, which is positioned on the right side of the screen.

This form includes the following components:

- **Language Selector**: This feature allows users to choose their preferred language, with the default set to English.
- **Full Name Input Field**: The initial field in the form is for entering the full name, labeled "**Full name**" with a red asterisk (*) to indicate it is mandatory.
- **Store Name Input Field**: The field in the form is for input Name of Store.

- **Email Input Field**: The field labeled "**Email**" with a red asterisk (*) indicates it is a necessary field. An example email address is displayed within this field.
- **Password Input Field**: This field, labeled "**Password**" with a red asterisk (*), is for entering the password. The input is masked for security, and there is an icon to toggle the visibility of the password.
- **Confirm Password Input Field**: This field, labeled "**Confirm password**" with a red asterisk (*), is for re-entering the password to ensure accuracy. It also includes an icon to show or hide the password.
- **Register Button**: Users click this button to submit their registration details to the server.

3.4.1.3 Material Management Page

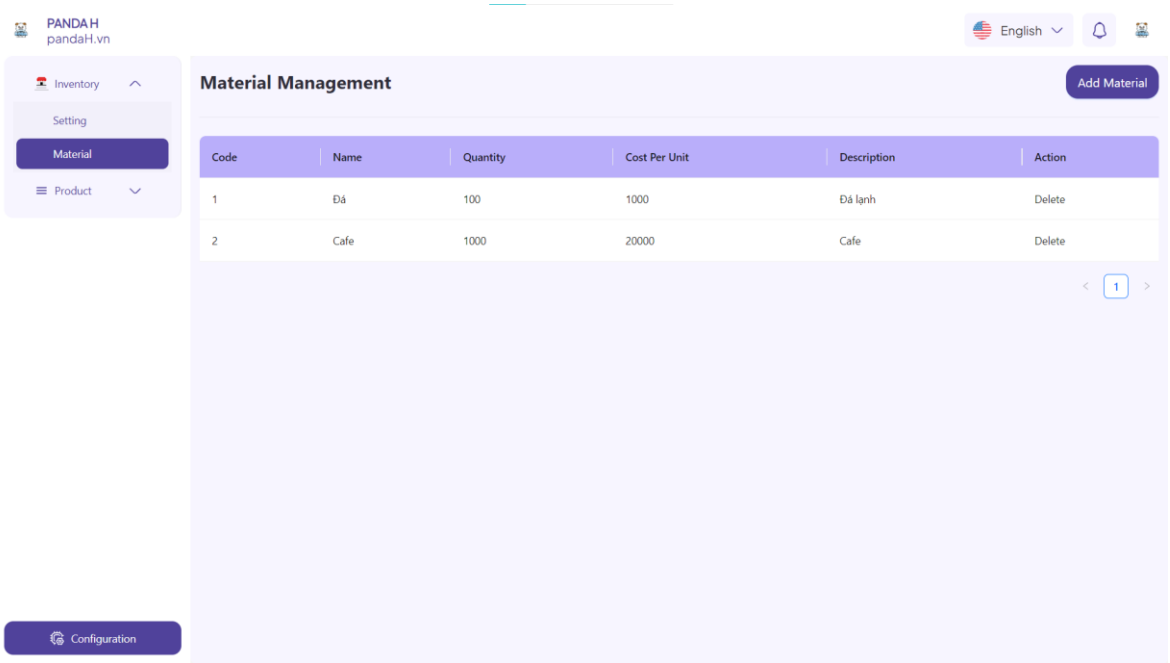


Figure 82. Material Management page

Material management page description:

The Materials Management page is part of the Admin system, designed to effectively manage material inventory.

I design minimalist and user-friendly interfaces, focusing on functionality and ease of use.

Below is a detailed description of the page:

- **Header section**: The top part of the page includes the PANDAH logo and website URL (pandaH.vn). On the right there is a language selector that allows users to select their preferred language, with the default being English. Next to the language selector is a notification bell icon.
- **Sidebar navigation**: On the left side of the screen, there is the sidebar navigation menu. It includes expandable sections for "**Inventory**" and "**Products**" management. In the Inventory section, with the latter selected to represent the current view.
- **Main content area**: The main content area is titled "**Document Management**", which clearly states the purpose of the page. The materials management form is located on the right side of the sidebar.
- **Material table**: The table is divided into several columns: "**Code**", "**Name**", "**Quantity**", "**Cost per unit**", "**Description**", and "**Action**".
    - The "**Code**" column displays a unique identification number for each material.
    - The "**Name**" column lists the name of the ingredient (e.g., Ice, Coffee).
    - The "**Quantity**" column displays the quantity of each ingredient available.
    - The "**Cost per unit**" column shows the cost of each unit of material.
    - The "**Description**" column provides additional information about the material.
    - The "**Action**" column includes a "**Delete**" button for each document, allowing users to remove the material from the list.

- **Add Material Button**: Located in the top right corner above the board, there is a prominent purple "**Add Material**" button. Users can click this button to add new materials to the inventory.
- **Pagination controls**: At the bottom right of the table, there are pagination controls that allow users to navigate through multiple material pages.

3.4.1.4 Create Material Page

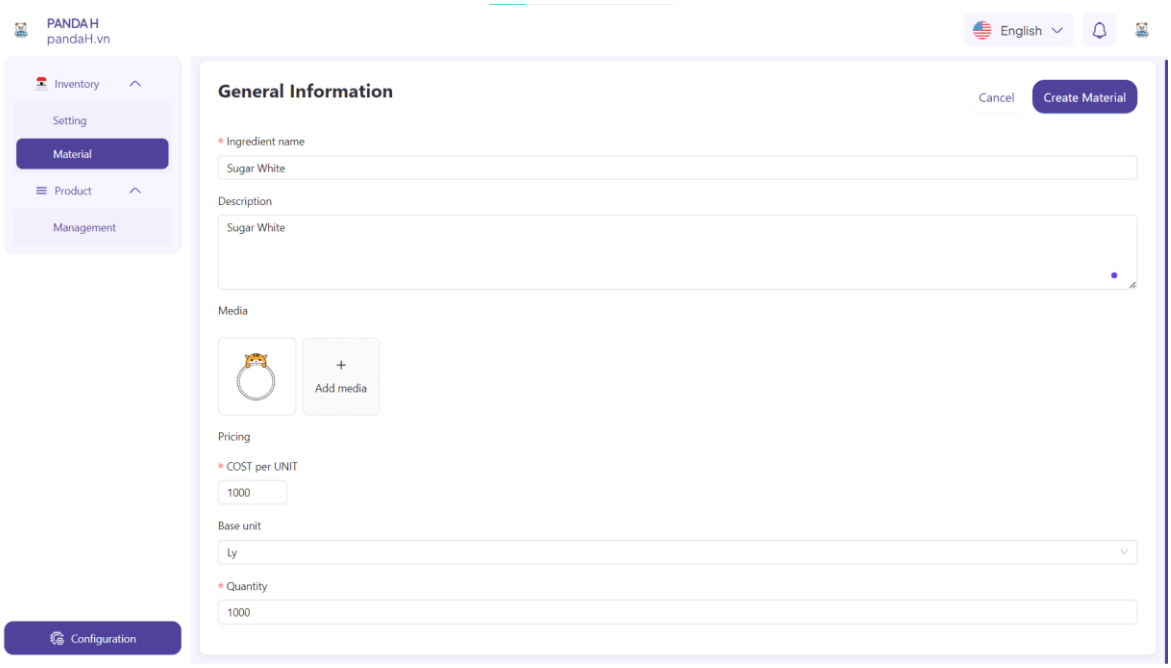


Figure 83. Material creation page

Material creation page description:

The Create Materials page is designed to facilitate the addition of new materials to the inventory system. The layout is simple and focuses on the information needed to create a new document entry.

Below is a detailed description of the page:

- **Ingredient Name Field**: The first field labeled "**Ingredidient name**" has a red asterisk (*) indicating it is a required field. The user enters the ingredient name here (for example: Sugar White).

- **Description Field**: Below the ingredient name is a "Description" field where users can provide additional information about the ingredient.
- **Media section**: This section allows users to add images or other media related to the document.
- **Cost per unit**: This field, labeled "**COST per UNIT**" with a red asterisk (*), is where the user enters the cost of one unit of material (for example, 1000).
- **Base unit**: The user specifies the base unit of measure for the material from drop down list. If store don't have base unit, user can create new Unit (e.g. Ly).
-

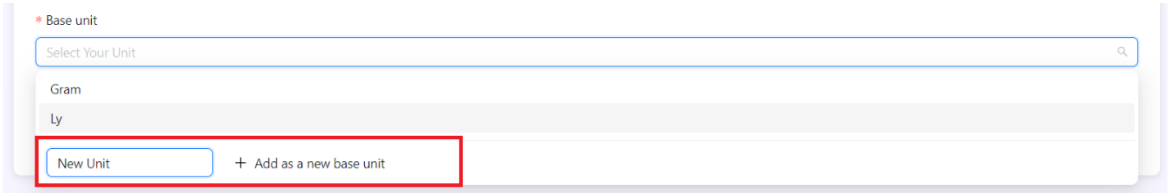


Figure 84. Add new unit example

- **Quantity Field**: This required field, labeled "Quantity" with a red asterisk (*), requires the user to enter the total quantity of ingredients available (for example, 1000).
- **Create Material Button**: Located in the top right corner of the form, the "**Create Material**" button allows users to submit the form and add new materials to the inventory.
- **Cancel Button**: Located next to the "Create Material" button, the "Cancel" button allows the user to discard any information entered and return to Material Management page.

3.4.1.5 Product Management Page

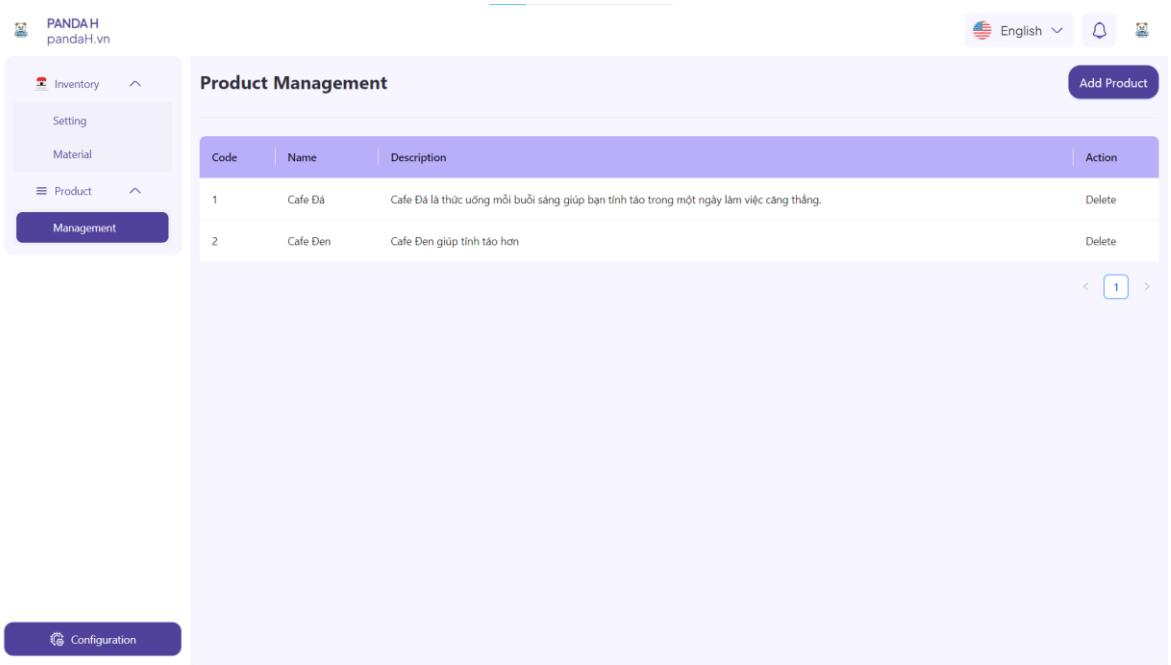


Figure 85. Product Management page

Product Management description page:

- Product table: The table is divided into several columns: "**Code**", "**Name**", "**Description**", and "**Action**".
    - The "**Code**" column displays a unique identification number for each product.
    - The "**Name**" column lists the name of the product (e.g., Cafe, …).
    - The "**Description**" column provides additional information about the product.
    - The "**Action**" column includes a "**Delete**" button for each document, allowing users to remove the product from the list.
    - **Add Material Button:** Located in the top right corner above the board, there is a prominent purple "Add Product" button. Users can click this button to add new products to the product.
    - **Pagination controls:** At the bottom right of the table, there are pagination controls that allow users to navigate through multiple product pages.

3.4.1.6 Add Product Page

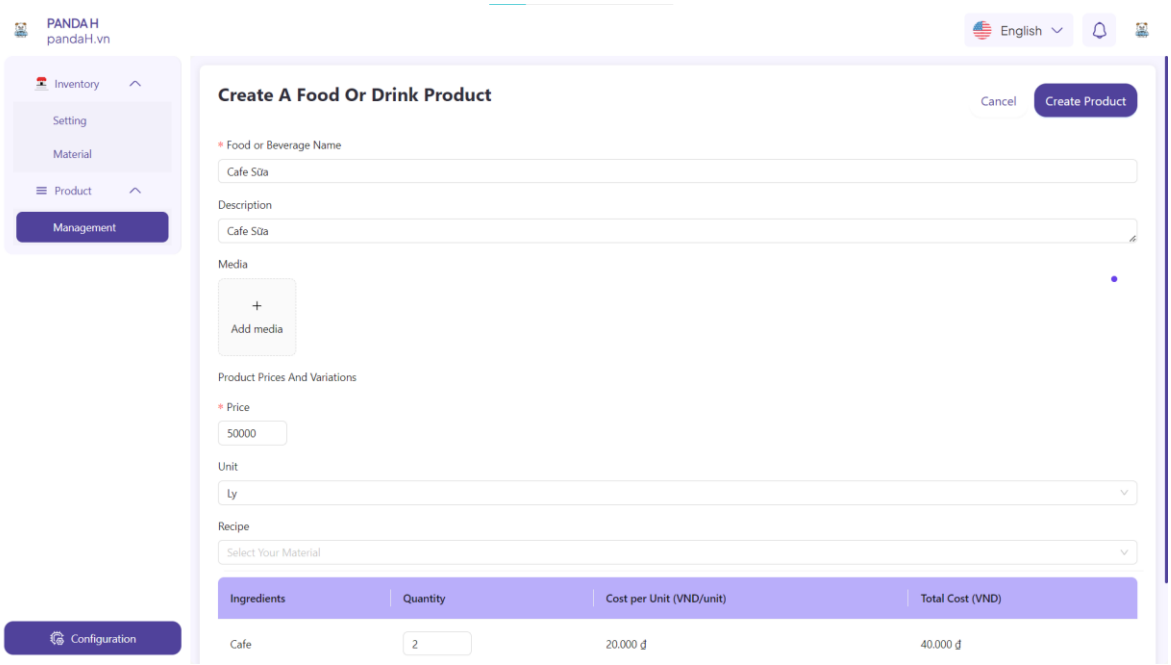


Figure 86. Create product page

Product creation page description:

The Create Products page is designed to facilitate the addition of new products to the product system. The layout is simple and focuses on the information needed to create a product entry.

Below is a detailed description of the page:

- **Food or Beverage Name Field:** The first field labeled " **Food or Beverage Name** " has a red asterisk (*) indicating it is a required field. The user enters the product name here (for example: Cafe Sữa).
- **Description Field:** Below the ingredient name is a "Description" field where users can provide additional information about the product.
- **Media section:** This section allows users to add images of product.
- **Price**: This field, labeled "Price" with a red asterisk (*), is where the user enters the cost of one unit of product (for example, 50000).

- **Base unit:** The user specifies the base unit of measure for the material from drop down list. If store don’t have base unit, user can create new Unit (e.g. Ly).

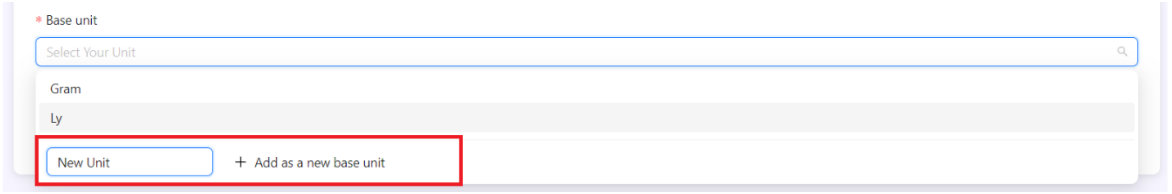


Figure 87. Add new unit example

- **Receipt**: This required field, user can select Material and field cost of unit for one product on a below table.

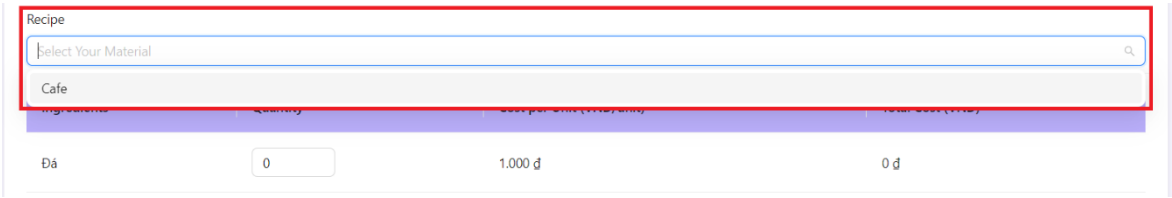


Figure 88. Add Recipe component

- **Create Product Button:** Located in the top right corner of the form, the "**Create**" button allows users to submit the form and add new products to the database.
- **Cancel Button:** Located next to the "Create" button, the "Cancel" button allows the user to discard any information entered and return to Product Management page.

3.4.2 Admin Web System

This Part was handling on Industry Experience Requirement, so I will show some case and capture of Product.

3.4.2.1 Login Page

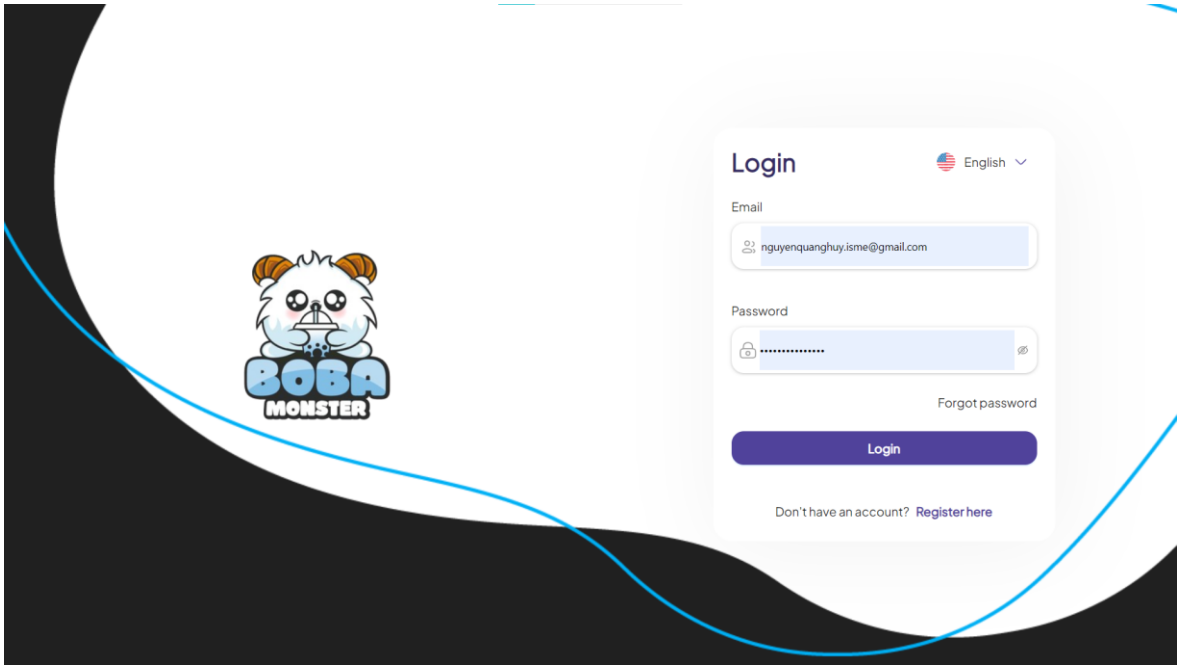


Figure 89. Admin login page

3.4.2.2 Admin Register Page

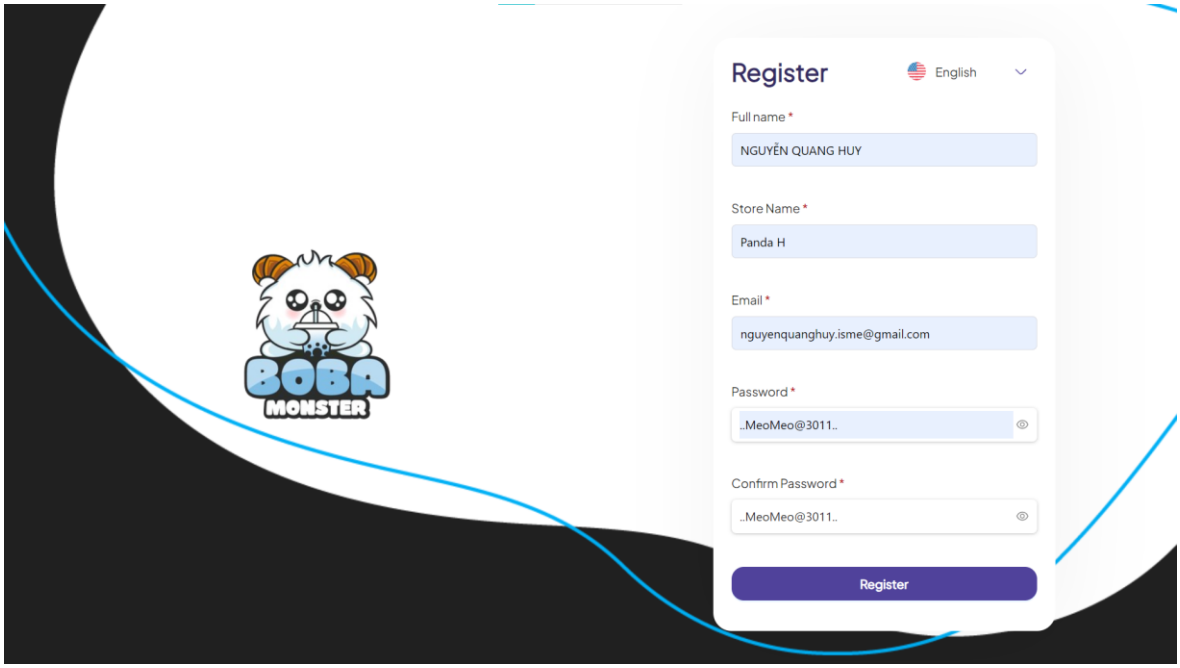


Figure 90. Admin register page.

3.4.2.3 Material Management Page

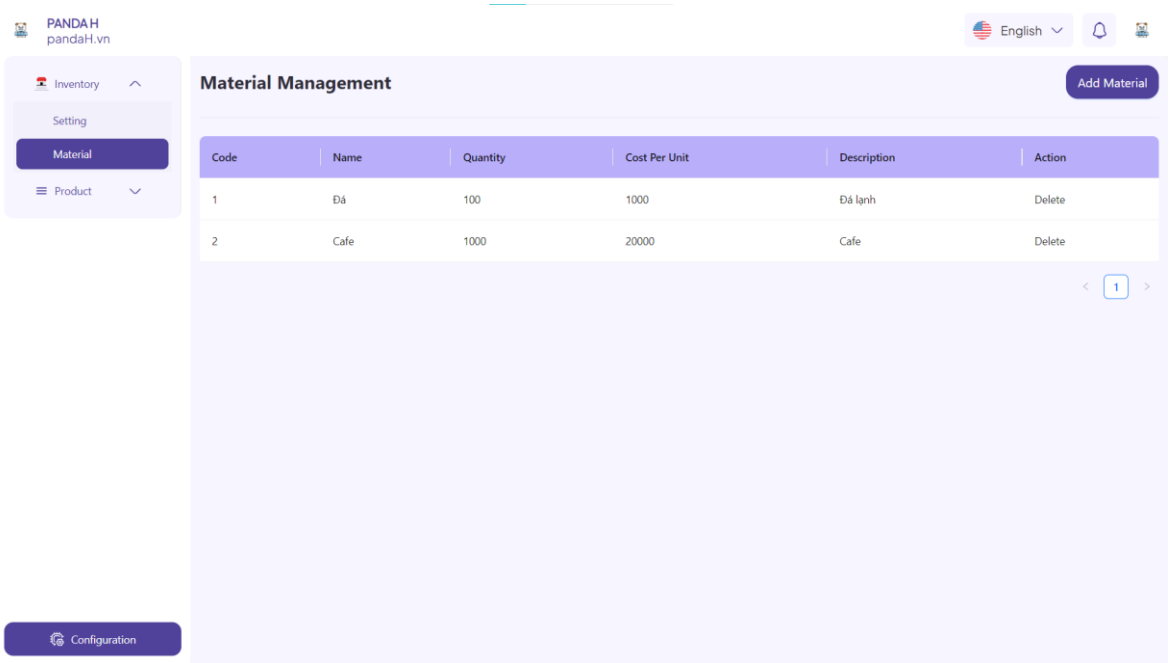


Figure 91. Material Management page

3.4.2.4 Create Material Page

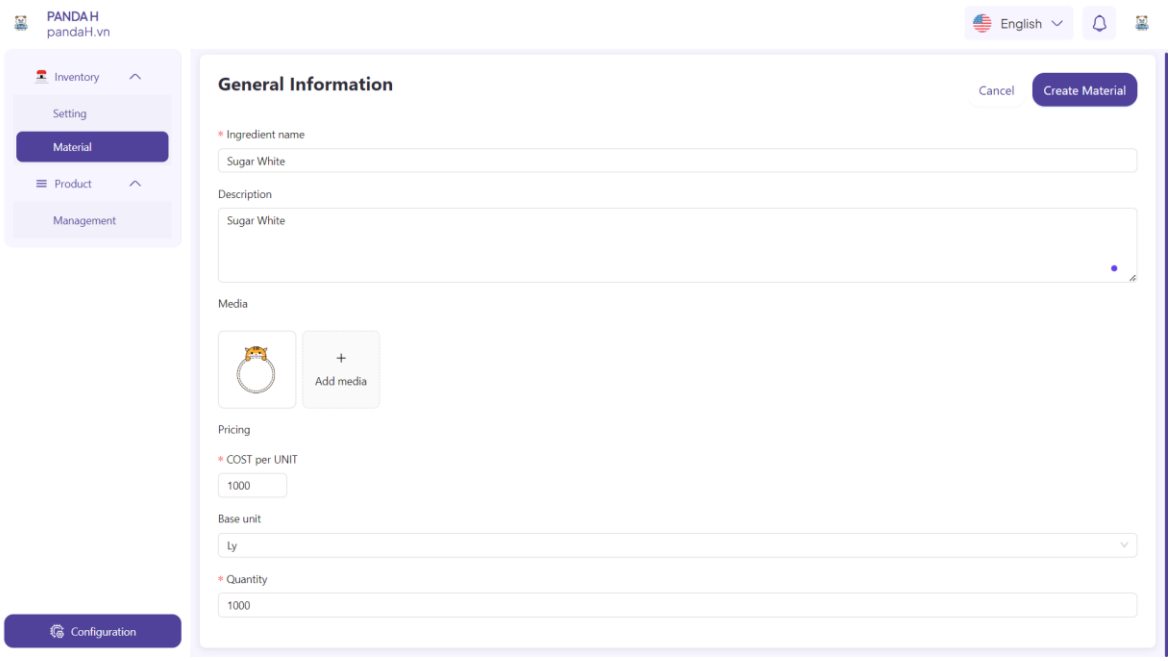


Figure 92. Material creation page

3.4.2.5 Product Management Page

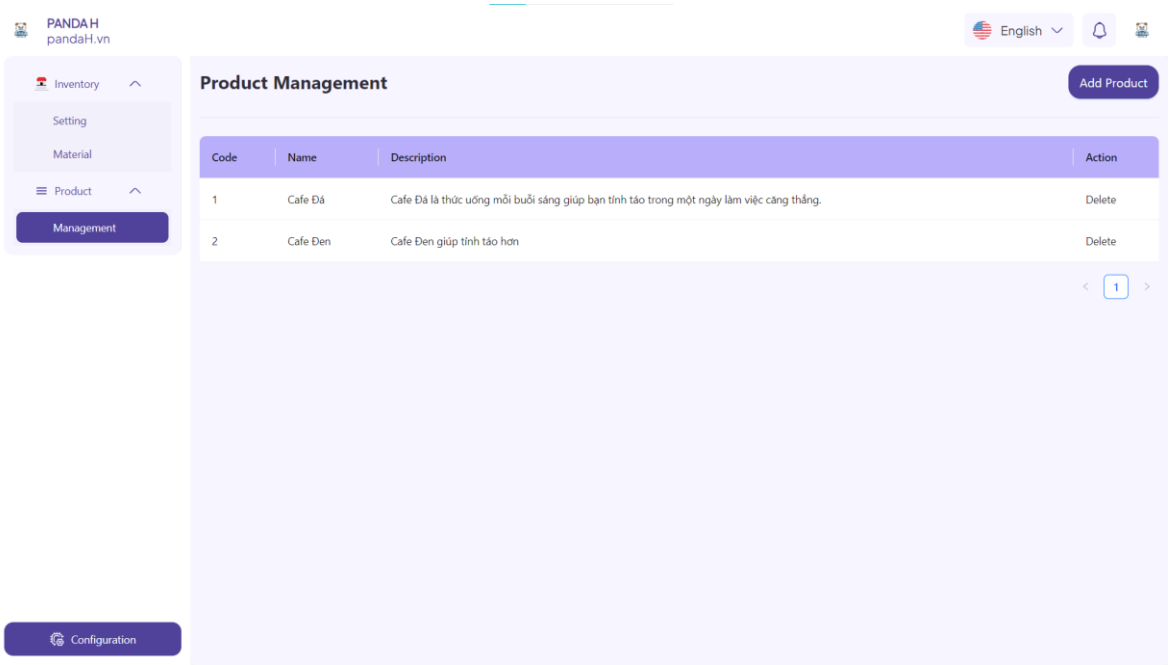


Figure 93. Product Management page

3.4.2.6 Add Product Page

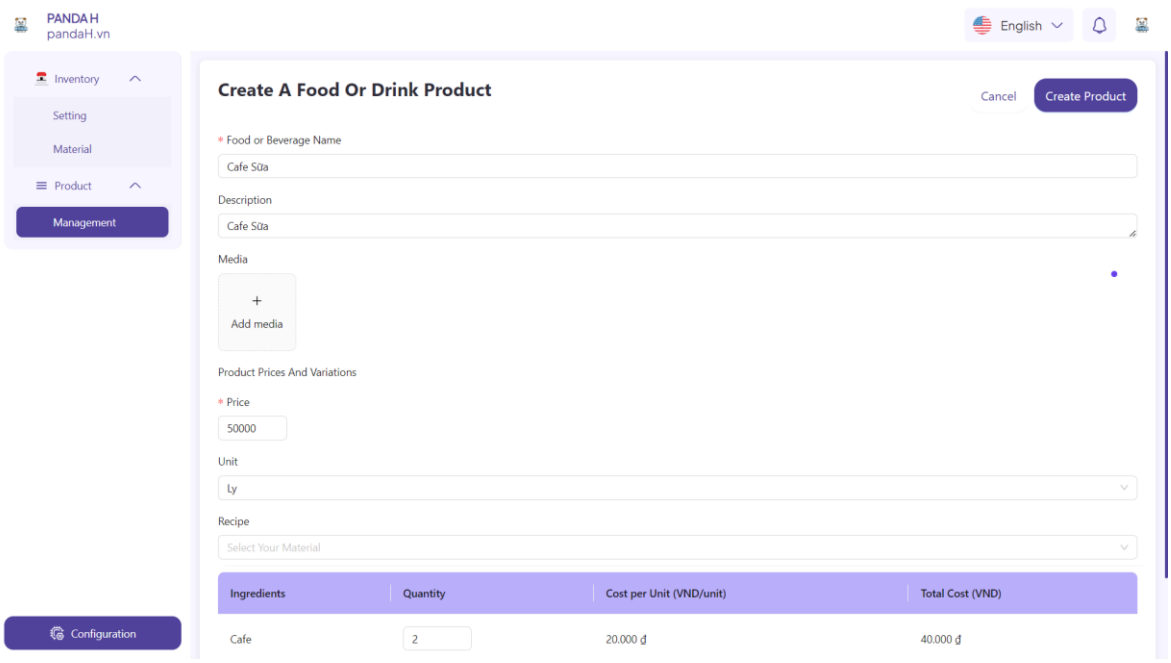


Figure 94. Create product page

3.4.3 POS Web App

3.4.3.1 Login Page

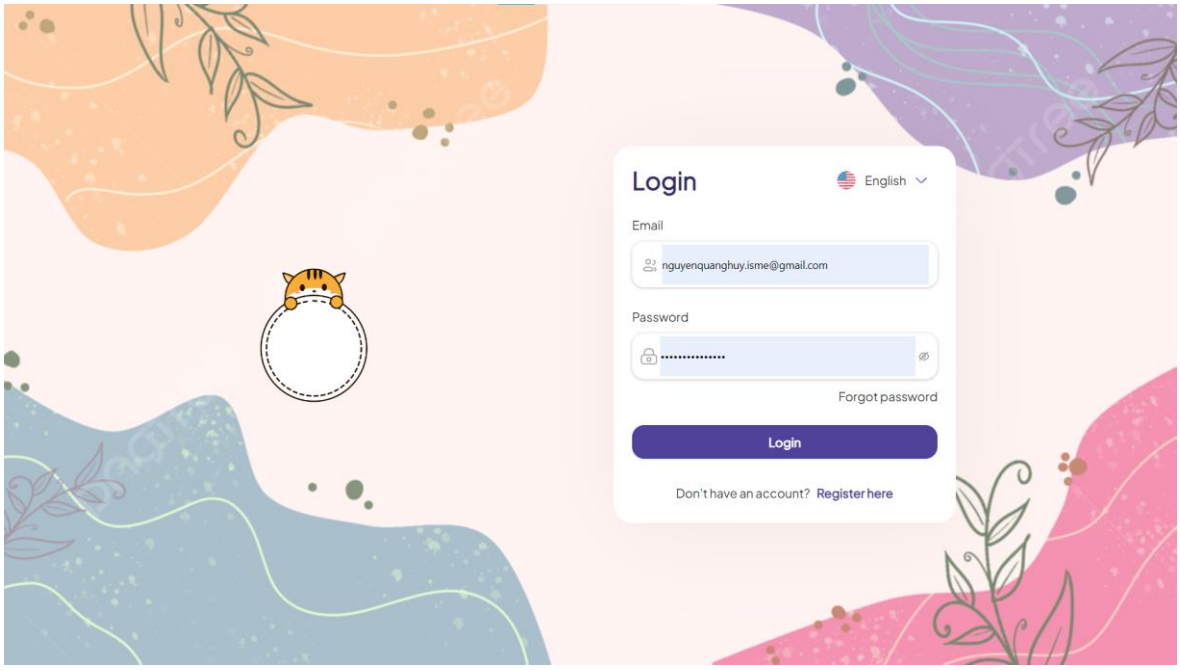


Figure 95. POS login page

Login page description:

The login page has backgrounds and in the middle of the page there is a login form placed on the right side.

The form includes the following elements:

- **Language selector**: In the top right corner of the form there is an option to select a language. The default language displayed is English ("English or Vietnamese"). This function is used to convert languages based on locales en/vi files.
- **Email input field**: Text field to enter email addresses.
- **Password input field:** Password input field for login function and this field has the function of showing or hiding the password.
- **Login button**: When the user enters username and password, they will press this button to authenticate and log in
- **Registration link**: When the store owner/admin does not have an account to configure the admin page, it can be used to register store information and other information.

3.4.3.2 Register Page

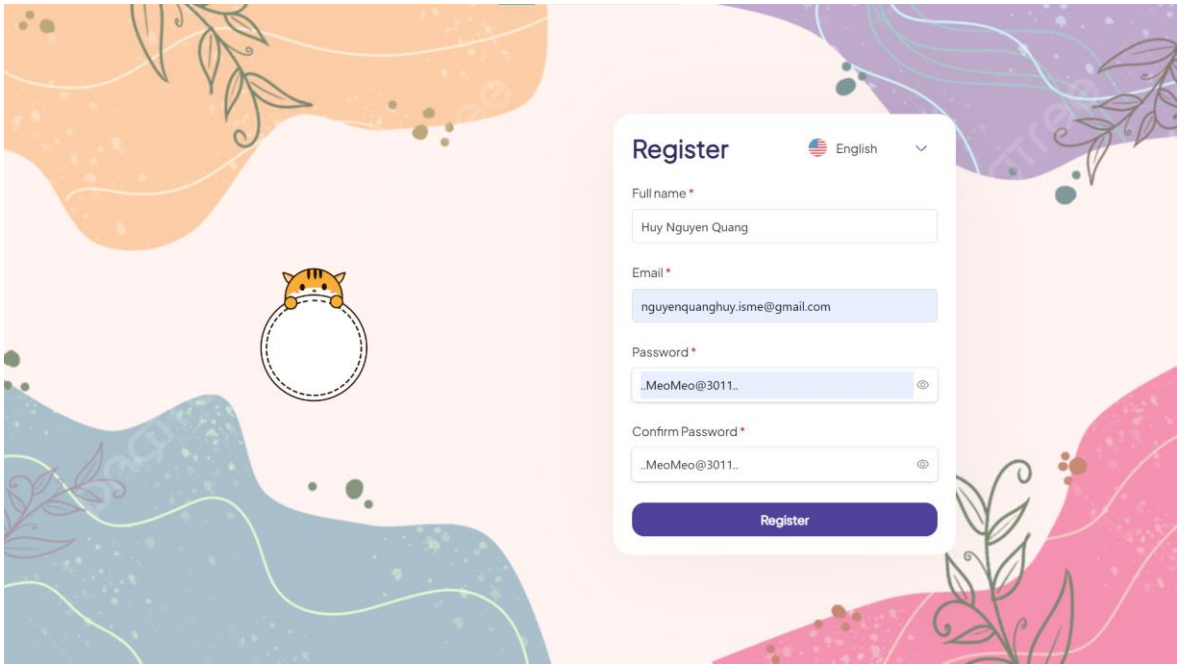


Figure 96. POS Register page

Register page description:

The registration page is designed with the focus of the page being the registration form, placed on the right side of the screen.

The form includes the following elements:

- **Language selector**: language selector allows users to select their preferred language. The default display language is English.
- **Full name input field**: The first field in the form is for entering the full name. It is labeled "Full name" with a red asterisk (*) indicating it is a required field.
- **Email input field**: The email input field, labeled "Email" with a red asterisk (*), indicates it is a required field. My email address is shown in this field as an example.

- **Password input field**: password input field, labeled "Password" with a red asterisk (*). The password is masked for security purposes, and there is an icon to the right of the input box to show or hide the password.
- **Password input field**: re-enter password field, this field is labeled "**Confirm password**" with a red asterisk (*). It ensures that users re-enter their passwords correctly. This field also includes an icon to show or hide the password.
- **Register button**: User clicks to submit their registration information to the server.

3.4.3.3 POS Home Page

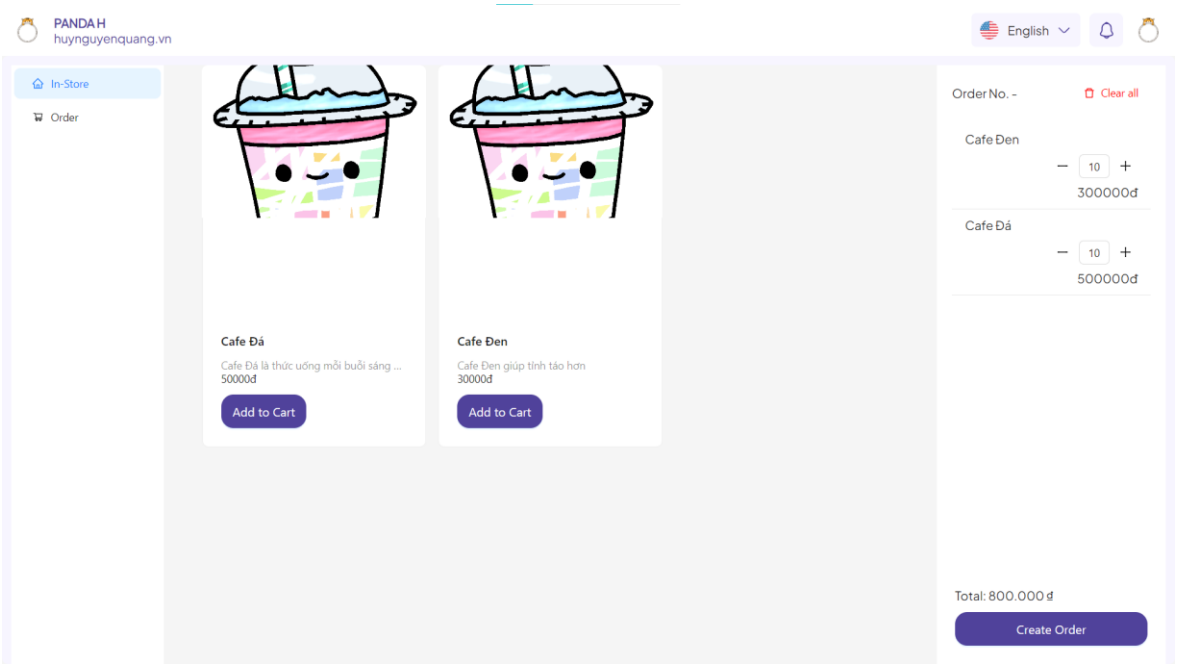


Figure 97. POS Home page

Order page description: The Order page is designed to support the order creation process in the POS system. I designed it to have a clean and intuitive interface, focusing on product selection and order creation.

Below is a detailed description of the page:

- **Header section**: At the top of the page displays the logo and website URL (hunguyenquang.vn). On the right, there is a language selector that allows users to select language, with the default being English.
- **Sidebar navigation**: On the left side, the sidebar navigation menu includes options to manage "In Store" and "Order". The "In Storer" option is home page of POS system.
- **Main content area**: The central part of the page is dedicated to displaying available products and managing current orders.
    - **List of products**:
        - Each **product** is displayed as a card with an image, name, description and price. For example, the page displays 2 products: "Iced Coffee" and "Black Coffee".
        - Each product card includes a brief description and price of the product (e.g. 300000 VND).
        - "**Add to Cart**" button below each product description that allows cashier to add items to cart.
    - **Order summary**: On the right side of the page there is an order summary.
        - **Order Number**: Shows the current order number with the option to delete all items in the cart by clicking "Delete All".
        - **Items in cart:** Lists items added to cart, including product name, quantity, and total cost for each item. For example: Café Đen: Quantity 10, Total 3000000 VND and Café Đá: Quantity 10, Total 5000000 VND
        - Users can adjust the quantity of each item using the plus and minus buttons.
        - **Total cost:** At the bottom of the order summary shows the total cost of the order (e.g. 8000000 VND).

- **Create Order Button**: Users/ Cashier can click this button to complete and submit the order.

3.4.3.4 POS Order List

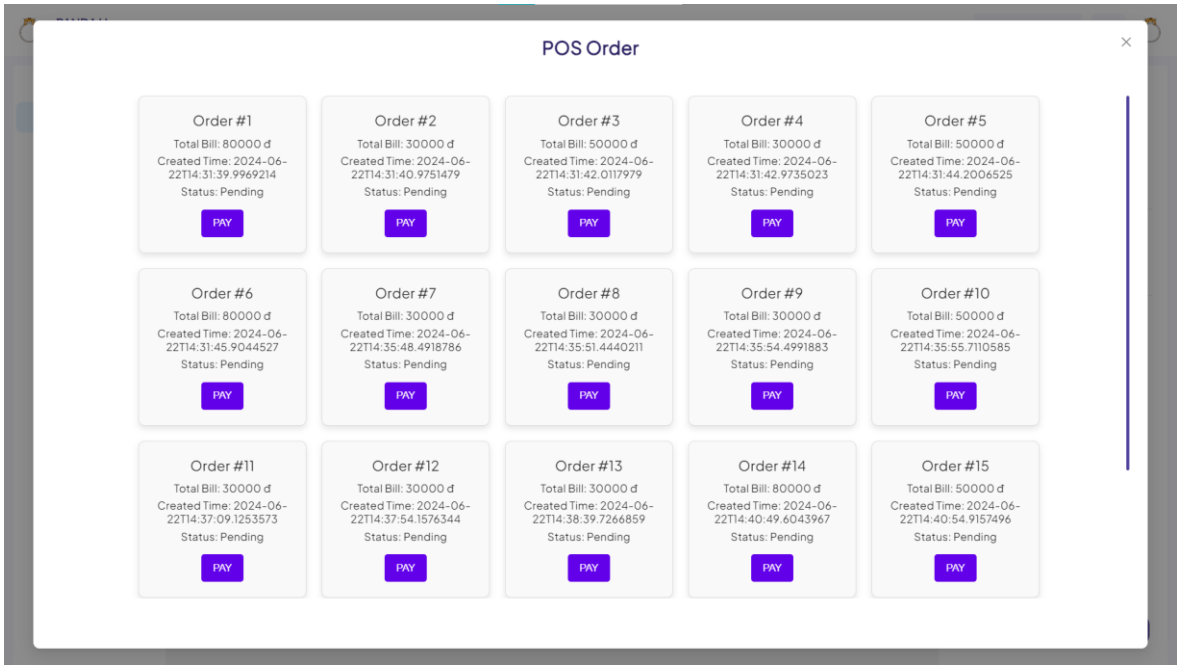


Figure 98. Order list page

Order List Page description: The order detail page supports users/cashiers to review created orders and perform the next payment steps (coming soon).

The Order Card include: Order No, Total Bill, Created Time and Status.