PROJECT FINAL DOCUMENTATION

CSE434 - Aspect and Service-Oriented Software Systems

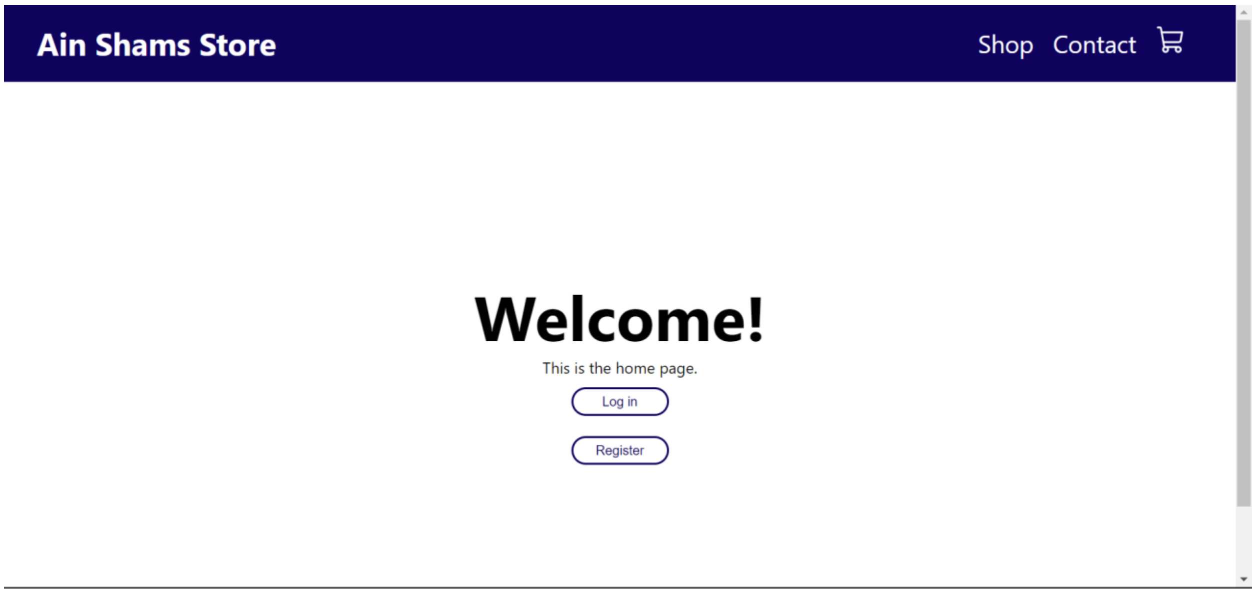
Ali Hesham Elgazzar 18P7355

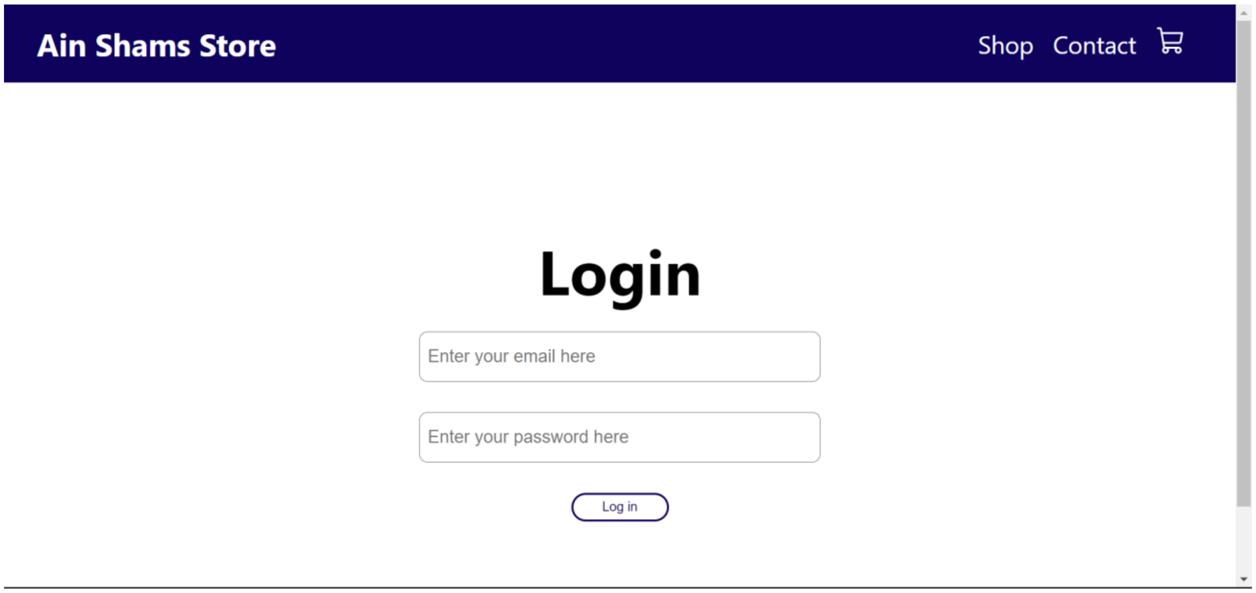
Ali Mohamed 19P3462

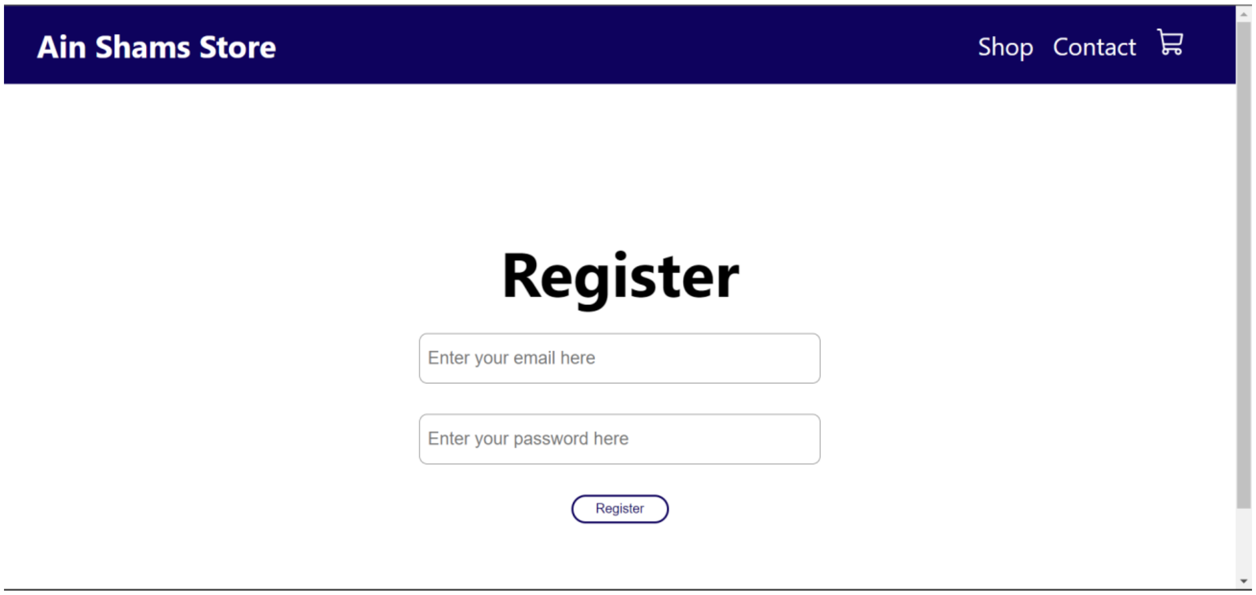
Adham Ehab 19P4388

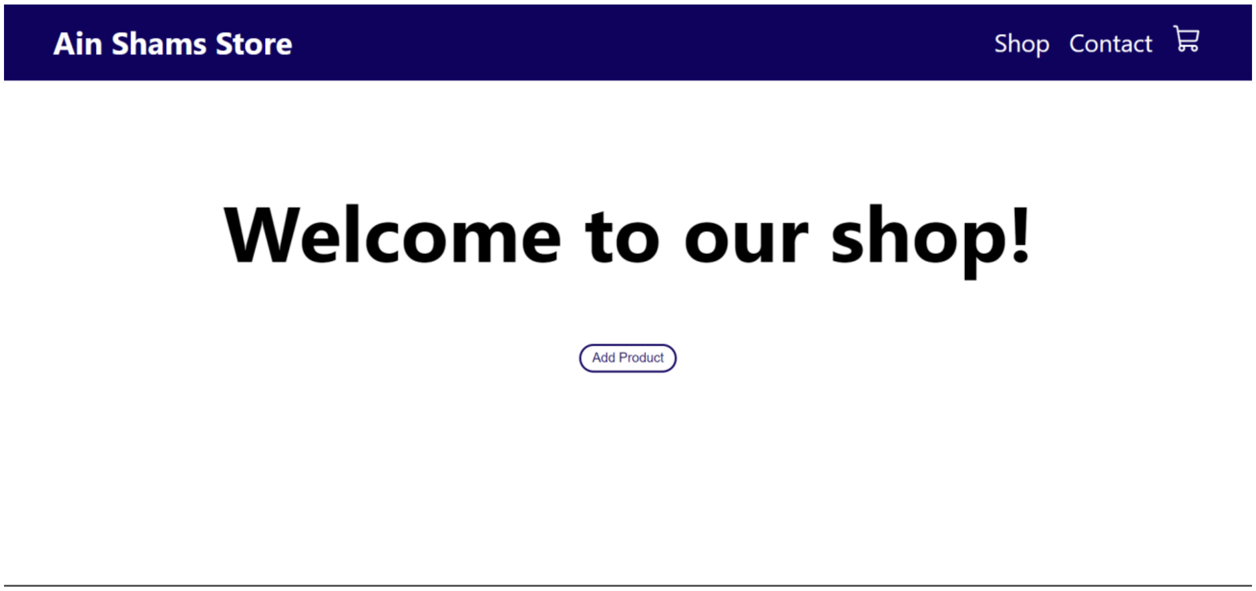
Basel Mohamed Ramadan 18P8121

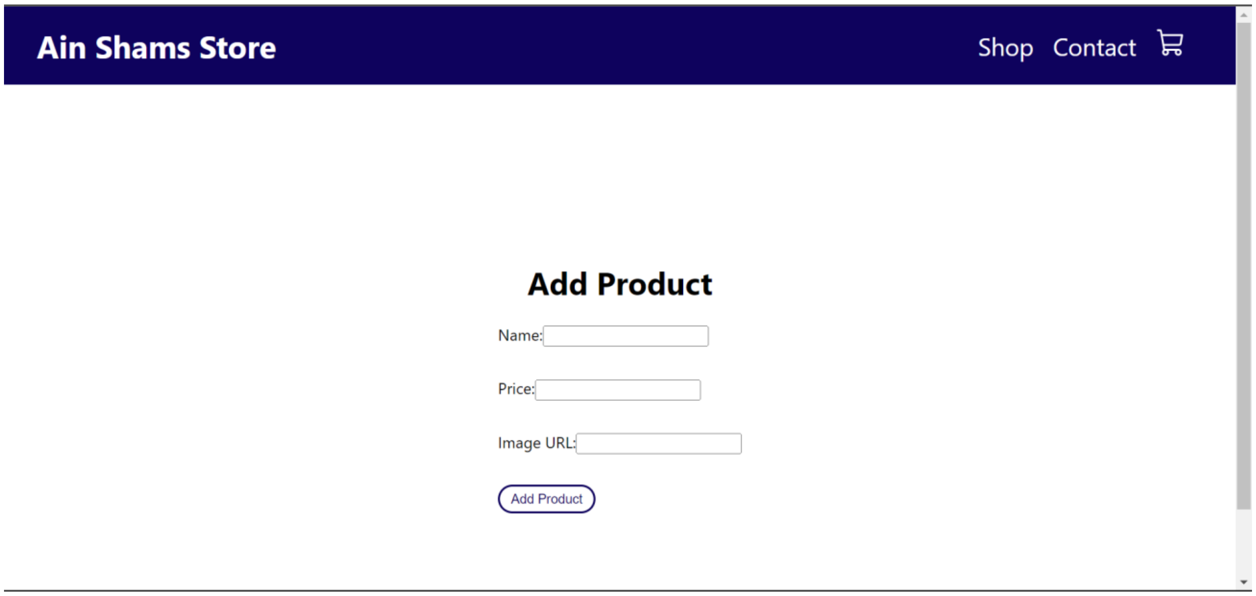
Application UI:



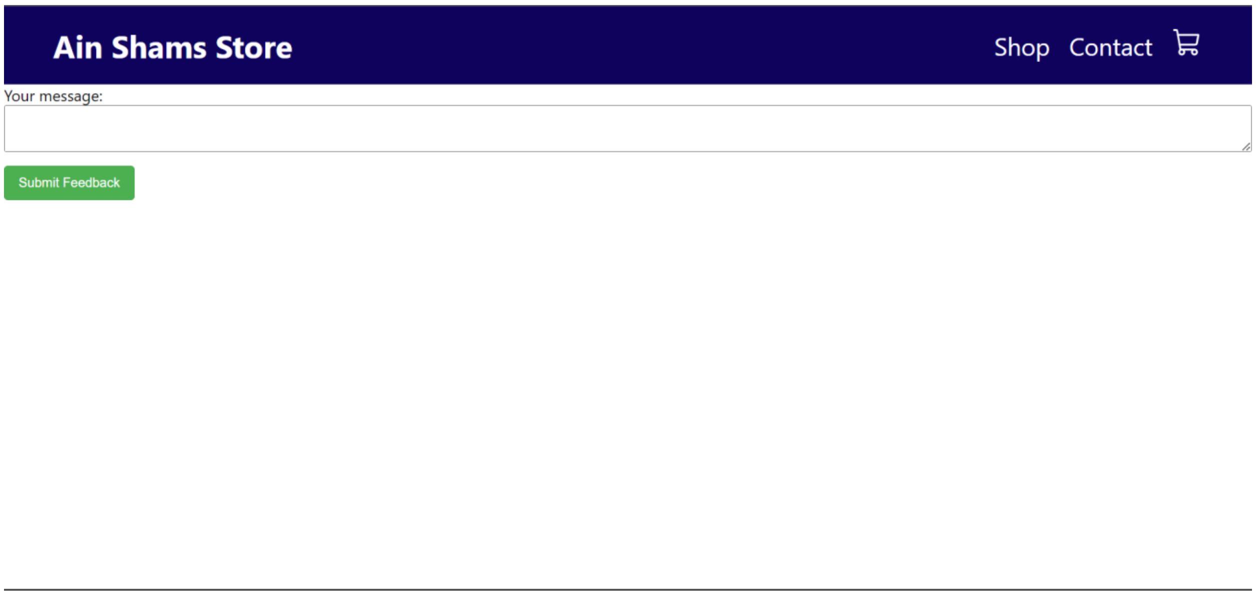



Ain Shams Store
Shop
Contact
Register
Enter your email here
Enter your password here
Register

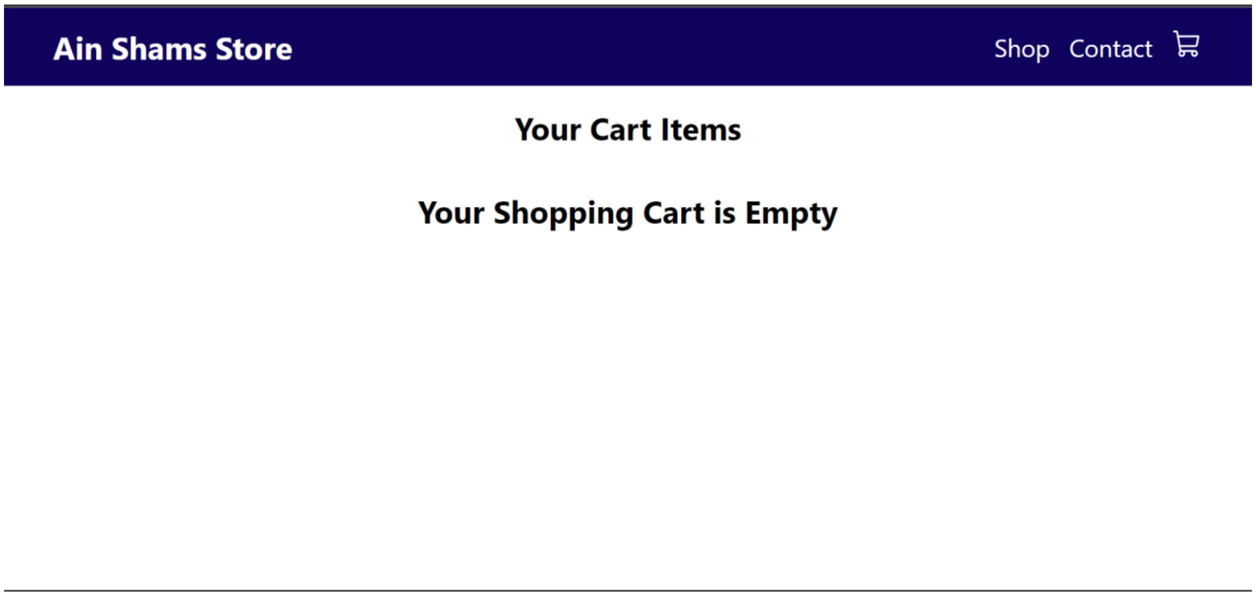

Ain Shams Store
Shop
Contact
Welcome to our shop!
Add Product



Ain Shams Store
Shop
Contact
Add Product
Name:
Price:
Image URL:
Add Product



Ain Shams Store
Shop
Contact
Your message:
Submit Feedback



Your Cart Items

Your Shopping Cart is Empty

1. Introduction

The application utilizes React for building user interfaces and incorporates various libraries and components to create a dynamic web application.

2. Code Overview

2.1. Import Statements

The code begins with import statements that bring in external dependencies, components, and stylesheets. Notable imports include React Router components (BrowserRouter, Routes, Route), custom components (Navbar, Shop, Contact, Cart), and necessary React hooks (useEffect, useState).

```
import "./App.css";

import { BrowserRouter as Router, Routes, Route } from "react-router-dom";

import { Navbar } from "./components/navbar";

import { Shop } from "./pages/shop/shop";

import { Contact } from "./pages/contact";

import { Cart } from "./pages/cart/cart";

import { ShopContextProvider } from "./context/shop-context";

import { useEffect, useState } from "react";

import Home from "./pages/home";

import Login from "./pages/login";

import Register from "./pages/register";
import AddProductForm from "./pages/addProductForm";
```

2.2. State Management

The application employs React hooks (useState) to manage state variables. Two state variables, loggedIn and email, are utilized to track the user's authentication status and email, respectively.

```
const [loggedIn, setLoggedIn] = useState(false);

const [email, setEmail] = useState("");
```

2.3. Routing

Routing is achieved using the react-router-dom library. The <Router> component wraps the entire application, and <Routes> contains individual <Route> components for different pages. Each <Route> specifies a path and an element prop, determining which component to render for a given route.

```
<Router>

  <Navbar />

  <Routes>

    {/* ... */}

  </Routes>

</Router>
```

2.4. Context Provider

The ShopContextProvider is used to provide context for the application, facilitating state management and communication between components. It wraps the entire application, allowing components to consume the shared context.

```
<ShopContextProvider>
  {/* ... */}
</ShopContextProvider>
```

2.5. Component Rendering

Components are rendered within the main <div> element with the class name "App." The Navbar component is placed outside the <Routes> to ensure its presence on all pages.

```
<div className="App">
  {/* ... */}
</div>
```

2.6. Pages

Various pages, such as Home, AddProductForm, Login, Register, Contact, Cart, and Shop, are specified within the <Routes> component and are rendered based on the current route.

```
<Routes>
  <Route path="/" element={<Home email={email} loggedIn={loggedIn} setLoggedIn={setLoggedIn} />} />
  {/* ... */}
</Routes>
```

2.7. Login and Registration

The Login and Register components receive functions (setLoggedIn and setEmail) as props, enabling them to update the state in the parent component (App), reflecting changes in authentication status and user email.

```
<Route path="/login" element={<Login setLoggedIn={setLoggedIn} setEmail={setEmail} />} />
<Route path="/register" element={<Register setLoggedIn={setLoggedIn} setEmail={setEmail} />} />
```

3. Conclusion

In conclusion, the React application exhibits a well-organized structure, utilizing React Router for navigation, context providers for state management, and various components to compose different pages. The state management through React hooks enhances the interactivity of the application, and the routing system ensures a smooth user experience. The code adheres to best practices in React development, promoting maintainability and scalability.

External Services:

FormsPree for email suggestions handling

The form in the provided React component utilizes a service called FormsPree for handling form submissions. FormsPree is an online form backend service that allows developers to set up HTML forms on their websites without the need for a server-side script. It simplifies the process of collecting form data by providing a seamless and easy-to-use solution.

Form Submission Handling:

FormsPree acts as a form endpoint where the data submitted through the form is sent.

Developers can specify the FormsPree endpoint as the form's action attribute, as seen in the code: action="https://formspree.io/f/mleqrzeq".

The method attribute is set to "POST," indicating that the form data will be sent to FormsPree using the HTTP POST method.

No Backend Setup Required:

One of the main advantages of FormsPree is that it eliminates the need for developers to set up a backend server to handle form submissions.

This is particularly beneficial for static websites or projects hosted on platforms that don't support server-side scripting.

Email Notifications:

FormsPree provides email notifications to inform the website owner or administrator whenever a form is submitted.

This feature ensures that users' feedback or information is promptly delivered to the intended recipients.

Simple Integration:

Integrating FormsPree with a form is a straightforward process. Developers only need to specify the FormsPree endpoint in the form's HTML, as demonstrated in the provided code snippet.

Security:

FormsPree handles form submissions securely, providing protection against common form-related vulnerabilities.

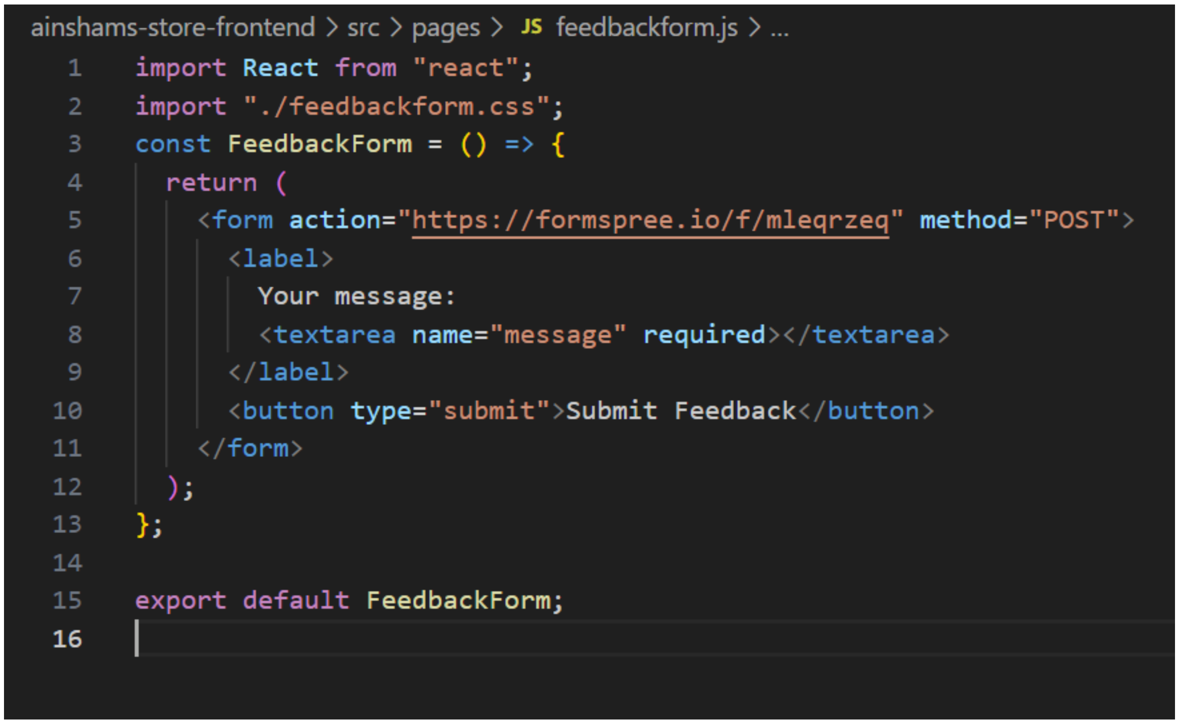
It helps prevent issues such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) by implementing security best practices.

Documentation and Support:

FormsPree provides documentation and support to help developers integrate and troubleshoot any issues.

The documentation typically includes examples and guidelines for setting up and customizing forms.

In summary, FormsPree is a user-friendly form backend service that simplifies the process of handling form submissions for developers, particularly those working on projects where setting up a server-side backend might be challenging or unnecessary. Its simplicity, ease of integration, and features like email notifications make it a popular choice for handling form submissions in web development.


ainshams-store-frontend > src > pages > JS feedbackform.js > ...

```
import React from "react";
import "./feedbackform.css";
const FeedbackForm = () => {
  return (
    <form action="https://formspree.io/f/mleqrzeq" method="POST">
      <label>
        Your message:
        <textarea name="message" required></textarea>
      </label>
      <button type="submit">Submit Feedback</button>
    </form>
  );
};

export default FeedbackForm;

```

Firebase:

Managed Service:

Firebase Authentication is a fully managed service provided by Google Firebase. It handles user authentication processes, including secure storage of user credentials, authentication state management, and session handling.

Security Features:

Firebase Authentication incorporates security features such as secure token generation, encryption, and protection against common authentication vulnerabilities.

Integration with Firebase Ecosystem:

Firebase Authentication seamlessly integrates with other Firebase services, allowing developers to build end-to-end solutions. For example, authenticated users can access Firebase Realtime Database or Cloud Firestore.

Scalability and Reliability:

As a cloud-based service, Firebase Authentication scales automatically to handle a growing number of users. It ensures the reliability and availability of authentication services.

User Management:

Developers can easily manage users through the Firebase Console or programmatically using Firebase Authentication SDKs. This includes actions like blocking users, resetting passwords, and managing user profiles.

Social Authentication:

Firebase Authentication supports various authentication methods, including email/password, phone number, and social identity providers such as Google, Facebook, Twitter, and more.

In summary, Firebase Authentication simplifies the implementation of user authentication in web and mobile applications. It provides a secure, scalable, and reliable service, abstracting away the complexities of authentication processes, and seamlessly integrates with other Firebase services within the broader Firebase ecosystem. Developers can focus on building features and rely on Firebase Authentication to handle user identity and security.

The code imports necessary modules from the Firebase Authentication library.

getAuth is a function used to create an instance of the Firebase Authentication service, and signInWithEmailAndPassword is a function specifically for email and password authentication.

firebaseApp contains the configuration for the Firebase app and is used to initialize Firebase services.

The getAuth function is called with the firebaseApp configuration, creating an instance of the Firebase Authentication service (auth).

This auth object is now used to interact with Firebase Authentication methods, such as signing in users.

handleSignIn is an asynchronous function that is triggered when the user clicks the "Log in" button.

Inside this function, the signInWithEmailAndPassword method is called on the auth object. It takes the email and password provided by the user.

The method returns a userCredential upon successful authentication.

If authentication is successful, the user is logged in, and the application navigates to the '/shop' route. The user's information can be accessed through userCredential.user.

If there is an error during the authentication process, an error message is logged to the console.

Backend:

AinshamsStoreBackendApplication.java

"AinshamsStoreBackendApplication." This class includes the main method, which serves as the entry point for the application. Annotated with @SpringBootApplication, the class signifies the primary configuration class for Spring Boot. The SpringApplication.run method initiates the Spring Boot application, triggering the auto-configuration process and launching the embedded web server. This application is designed to serve as the backend for the Ainshams Store project, providing a foundation for building and deploying Java-based web services using the Spring Boot framework.

Product.java

Product class annotated with @Entity, indicating that instances of this class will be entities that can be stored in a relational database. The class represents a product entity with attributes such as id, name, price, and image.

The @Id annotation specifies that the id field is the primary key for the entity, and @GeneratedValue indicates that the value for this field will be generated automatically by the database upon insertion, using the GenerationType.IDENTITY strategy.

The class includes default and parameterized constructors for creating instances of the Product class. Getters and setters are provided for accessing and modifying the private fields. The name, price, and image attributes represent the product's name, price, and image, respectively.

This class is designed to be used in conjunction with a Java Persistence API (JPA) implementation, such as Hibernate, to enable seamless interaction with a relational database. Instances of this class can be persisted and retrieved from the database, making it suitable for managing product data in the context of a backend application for the Ainshams Store project.

ProductController.java

ProductController class, serving as a RESTful web controller for managing product-related operations in the Ainshams Store backend. Annotated with @RestController, it indicates that this class handles HTTP requests and produces JSON responses. The @CrossOrigin annotation allows cross-origin resource sharing (CORS) for requests from "http://localhost:3000."

The class is autowired with a ProductService instance to delegate business logic and data management operations to a service layer.

The controller exposes several REST endpoints:

Get All Products:

Endpoint: GET /products

Method: getAllProducts()

Returns a list of all products.

Create Product:

Endpoint: POST /products

Method: createProduct(@RequestBody Product product)

Creates a new product based on the provided request body.

Get Product by Id:

Endpoint: GET /products/{id}

Method: getProductById(@PathVariable Long id)

Retrieves a product by its ID. Returns a ResponseEntity with the product if found, or a not-found response if the product is not present.

Update Product:

Endpoint: PUT /products/{id}

Method: updateProduct(@PathVariable Long id, @RequestBody Product productDTO)

Updates an existing product with the given ID based on the information provided in the request body. Returns a ResponseEntity with the updated product if successful, or a not-found response if the product is not present.

Delete Product:

Endpoint: DELETE /products/{id}

Method: deleteProduct(@PathVariable Long id)

Deletes a product with the given ID. Returns a ResponseEntity with a success message if successful, or a not-found response if the product is not present.

These endpoints demonstrate typical CRUD (Create, Read, Update, Delete) operations for managing product entities. The exception handling in the controller ensures that appropriate responses are returned in case of errors or when attempting operations on non-existent products. The controller acts as an interface between the client (e.g., a front-end application) and the ProductService to handle product-related requests and responses.

ProductRepository.java

ProductRepository interface that extends the JpaRepository interface provided by Spring Data JPA. This interface serves as a repository for performing CRUD operations (Create, Read, Update, Delete) on Product entities in the Ainshams Store backend.

JPA Repository Interface:

The interface extends JpaRepository<Product, Long>.

Product is the entity type that the repository manages, and Long is the type of the entity's primary key (id).

By extending JpaRepository, this interface inherits a set of generic methods for basic CRUD operations and additional querying capabilities.

Automatic Implementation:

Spring Data JPA automatically generates implementations for the methods defined in this interface. Developers do not need to provide the implementation; Spring Data JPA handles it based on the naming conventions of the methods.

Generic Types:

Product represents the entity type managed by this repository.

Long is the type of the primary key of the Product entity.

CRUD Operations:

The repository provides methods for common CRUD operations such as save, findById, findAll, delete, etc.

For example, save is used to persist a new or updated product, and findById retrieves a product by its primary key.

Query Methods:

Spring Data JPA allows the creation of query methods based on method names. For example, a method named findByAttributeName would automatically generate a query to find entities based on the specified attribute.

Inherited Methods:

The JpaRepository interface inherits methods from the PagingAndSortingRepository and CrudRepository interfaces. This includes methods for pagination, sorting, and other common repository operations.

This repository interface acts as an abstraction layer that allows the service layer (e.g., ProductService) to interact with the database without having to deal with specific database operations. The Spring Data JPA framework provides a powerful and convenient way to handle data access and persistence in a Spring Boot application.

ProductService.java (Service)

ProductService class, annotated with @Service, indicating that it serves as a service component in the Ainshams Store backend. This service class encapsulates business logic related to products and acts as an intermediary between the ProductController and the ProductRepository.

Dependency Injection:

The ProductService is autowired with a ProductRepository instance to facilitate communication with the underlying database. Dependency injection is achieved through the @Autowired annotation.

CRUD Operations:

The service provides methods for common CRUD operations on products: getAllProducts, createProduct, getProductById, updateProduct, and deleteProduct.

getAllProducts retrieves a list of all products.

createProduct adds a new product to the database.

getProductById retrieves a product by its ID.

updateProduct updates an existing product based on the provided data.

deleteProduct removes a product from the database.

Business Logic:

In the updateProduct method, the service first checks if the product with the specified ID exists. If it does, the product attributes are updated with the values from the provided productDTO (Data Transfer Object), and the updated product is saved to the database.

Service Layer:

The service layer provides an abstraction over the repository, encapsulating business logic and promoting separation of concerns. It allows for more modular and maintainable code.

Overall, the ProductService plays a crucial role in managing the interaction between the ProductController and the database, ensuring that business logic is centralized and adhering to good design practices in a Spring Boot application.

Aspects:

Logging Aspect:

Aspect Annotation:

The @Aspect annotation marks the class as an aspect, allowing it to define cross-cutting concerns that can be applied to multiple parts of the application.

Component Annotation:

The @Component annotation indicates that the class is a Spring component and should be automatically registered in the Spring application context.

Advice Method:

The productMethods method is an advice method annotated with @After.

It is executed after the execution of methods matching the specified pointcut expression, which, in this case, targets methods in classes containing "Product" in their names within any package (*..*Product*.*(..)).

JoinPoint Parameter:

The JoinPoint parameter allows access to information about the intercepted method, such as method name, parameters, and target class.

Logging Action:

Inside the advice method, the method name is obtained from the JoinPoint using joinPoint.getSignature().getName().

A simple log statement is printed to the console, indicating that the identified action has been executed.

Pointcut Expression:

The pointcut expression (execution(* *..*Product*.*(..))) specifies the conditions under which the advice should be applied. It targets any method (*) in any class (*..*) that contains "Product" in its name.

Use Case:

This aspect is used to log method invocations related to entities or operations involving "Product" in their names. For example, it is used to log when product-related CRUD operations (update, delete) occur in the application.

Console Output:

When a method matching the specified conditions is executed, a log statement is printed to the console, providing information about the executed action.

This aspect demonstrates a basic form of logging cross-cutting concerns related to methods with "Product" in their names, offering a way to centralize and monitor certain actions in the application.

Exception Handling Aspect:

Aspect Annotation:

The @Aspect annotation marks the class as an aspect, indicating that it contains cross-cutting concerns that can be applied to specific methods.

Component Annotation:

The @Component annotation indicates that the class is a Spring component and should be automatically registered in the Spring application context.

Advice Method:

The beforeMyServiceMethods method is an advice method annotated with @After.

It is executed after the execution of methods matching the specified pointcut expression.

Pointcut Expression:

The pointcut expression targets two methods in the ProductService class: deleteProduct and updateProduct.

The expression uses execution(* com.example.ProductService.deleteProduct(..)) and execution(* com.example.ProductService.updateProduct(..)) to specify the exact methods.

Exception Handling:

Inside the advice method, an attempt is made to find a product by its ID using productRepo.findById(id).

If the product with the specified ID is not present (!optionalProduct.isPresent()), an exception is thrown with a message indicating that the product with the given ID is not found.

Use Case:

This aspect is used to perform pre-validation before executing the deleteProduct and updateProduct methods in the ProductService. The intention is to throw an exception if the specified product ID is not found, allowing for graceful error handling.

This aspect demonstrates a simple form of exception handling for specific methods in the ProductService, ensuring that if a product with a given ID is not found, an exception is thrown to indicate the error