ÇUKUROVA UNIVERSITY

ENGINEERING FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

GRADUATION THESIS

SUBJECT

Blog Supported E-commerce Site

BY

Amine Ceyda TANDOĞAN 2018555061

ADVISOR

Prof. Dr. Ayşe Selma ÖZEL

June 2023

ADANA

Abstract

This project is a comprehensive combination of an e-commerce platform and a blog site, offering a user-friendly and engaging experience for both visitors and registered members. The platform allows users to browse and review products without the need for membership, while also providing access to a range of informative blog posts. Upon logging in, members gain additional privileges, including the ability to make purchases and contribute their own blog posts, as well as manage and update their existing content. The project utilizes MySQL for the database management, Spring Boot for the backend development, and Angular for the frontend implementation, resulting in a seamless integration of technology to deliver a robust and dynamic platform. Through this project, the aim is to provide users with a comprehensive online shopping experience, complemented by valuable content and user-generated contributions, fostering engagement, trust, and long-term customer loyalty.

1. Introduction

In the digital era, the fusion of e-commerce and blogging has become increasingly prevalent, revolutionizing the way businesses connect with their customers and promote their products or services. This project focuses on creating a comprehensive platform that combines the functionalities of an e-commerce website and a blog site. Users will have the ability to explore and review products without membership, access informative blog posts, and after logging in, enjoy the full benefits of membership, including shopping capabilities and the ability to contribute, edit, and manage their own blog posts. The project utilizes MySQL for database management, Spring Boot for backend development, and Angular for frontend implementation, resulting in a seamless integration of these technologies. The aim of this project is to provide users with a captivating online shopping experience, complemented by valuable content and user-generated contributions,æ fostering engagement, trust, and long-term customer loyalty.

The e-commerce website component of the project is designed to offer a user-friendly interface where visitors can effortlessly browse and review various products. The platform will provide detailed product descriptions and high-quality images. By allowing users to explore products without the need for membership, the project aims to provide a convenient and accessible experience for all visitors.

In addition to the e-commerce functionality, the project incorporates a blog section that serves as an information hub for users. The blog will contain a wide range of articles, guides, and industry insights, offering valuable information and enhancing the overall user experience. Users can access this content without being a member, enabling them to benefit from the informative resources provided by the platform.

Once users choose to become members by logging in, they gain access to additional features and benefits. Members can not only make purchases but also contribute to the platform by adding their own blog posts. The project ensures that members have full control over their content, allowing them to edit, update, or delete their blog posts as desired. This feature empowers members to actively participate in the platform's community and share their insights, experiences, and expertise.

To achieve the seamless integration of the e-commerce and blog functionalities, the project utilizes MySQL for the database management. This allows for efficient storage, retrieval, and management of product data, blog posts, user information, and other relevant content. The Spring Boot framework is employed for the backend development, providing a robust and scalable foundation for the project. Angular is utilized for the frontend implementation, delivering a dynamic and intuitive user interface that enhances the overall user experience.

In conclusion, this project presents a comprehensive platform that combines the features of an e-commerce website and a blog site. By leveraging the power of MySQL, Spring Boot, and Angular, the project aims to provide a captivating and user-friendly experience for visitors

and members alike. The seamless integration of e-commerce and blogging functionalities, along with the ability for members to contribute their own content, creates an engaging and informative platform that fosters customer engagement, trust, and loyalty.

2. Related Work

In the rapidly evolving field of e-commerce and content marketing, there have been numerous studies and developments that shed light on the significance and potential of integrating an e-commerce website with blog pages. This section explores some of the current studies and similar systems/works that are relevant to the project/assignment.

E-commerce Platforms with Blog Integration:

Several e-commerce platforms have recognized the value of incorporating blog functionality into their systems. Platforms such as Shopify, WooCommerce, and Magento offer built-in blog features, allowing businesses to seamlessly combine their online store with content-rich blog pages. These platforms provide a range of templates, customization options, and content management systems that enable businesses to create engaging blog content alongside their product listings. The success of these platforms demonstrates the growing demand for integrated e-commerce and blogging solutions.

E-commerce Blogs as Customer Engagement Tools:

There is a growing body of research emphasizing the role of blogs in enhancing customer engagement and building brand loyalty. Blogs provide businesses with a platform to share their expertise, educate customers, and cultivate a sense of community. Studies have shown that consumers who engage with blog content are more likely to develop trust in the brand, make repeat purchases, and become brand advocates. By delivering informative and engaging content, businesses can establish themselves as thought leaders and forge stronger connections with their target audience.

User Experience and Navigation in E-commerce Websites:

Research on user experience and navigation in e-commerce websites is crucial to the success of this project/assignment. Studies have explored various aspects such as website layout, product categorization, search functionality, and checkout processes. Understanding user behavior, preferences, and pain points can help in designing an intuitive and user-friendly e-commerce website. Examining existing research on user experience in e-commerce will guide the project/assignment towards implementing best practices and ensuring a seamless shopping experience for customers.

In conclusion, the related work section highlights the current studies and similar systems/works that are relevant to the project/assignment. By examining e-commerce platforms with blog integration, content marketing and SEO research, the significance of blogs in customer engagement, and user experience studies in e-commerce websites, valuable insights can be gained to inform the design and implementation of the e-commerce website with blog pages. Building upon the knowledge and findings from existing research will

contribute to creating a robust and effective system that meets the needs and expectations of both businesses and customers.

3. Materials and Methods

In this section, the technologies, languages, tools, datasets, methods and research methods we used while conducting this study are given with their explanations and definitions.

3.1. Tools and Technologies Used In Project

3.1.1. Technology Stack

The project utilized a comprehensive technology stack comprising Spring Boot (Version 3.1.0) and Java (Version 17) for backend development. The choice of Spring Boot was driven by its simplicity, convention-over-configuration approach, and extensive ecosystem, while Java provided a robust and feature-rich programming language for enterprise-level applications. Maven was employed as the project's build and dependency management tool. On the frontend, Angular, TypeScript, HTML, CSS, Bootstrap, and FontAwesome were chosen to create a visually appealing and interactive user interface. MySQL served as the reliable and scalable database management system. Development tools such as IntelliJ IDEA, Visual Studio Code, Git, MySQL Workbench, Postman, Node.js, npm, and Tsc were integral to enhancing productivity, collaboration, and efficient development. This carefully selected technology stack ensured a streamlined development process and contributed to the successful implementation of the project.

3.1.2. Backend Development

In the backend development of my project, Spring Boot, version 3.1.0, along with Maven, was chosen as the primary framework and build tool. The backend was implemented using Java, version 17, which offers a robust and feature-rich programming language for enterprise-level applications.

1. Spring Boot Framework: Spring Boot was selected as the backend framework due to its simplicity, convention-over-configuration approach, and extensive ecosystem. It provides a streamlined development experience by eliminating the need for manual configuration, allowing developers to focus on writing business logic rather than boilerplate code. Spring Boot's auto-configuration feature greatly simplifies the setup of various components, such as database connections and web services, making it an ideal choice for rapid application development.
2. Maven: Maven serves as the project's build and dependency management tool. It simplifies the management of project dependencies, including Spring Boot and other libraries, by providing a declarative approach to specify and resolve dependencies.

Maven automates the process of building, packaging, and deploying the application, enhancing project organization and ensuring consistent builds across different environments.

3. Java 17: The project utilizes Java 17 as the programming language for backend development. Java offers strong support for object-oriented programming, scalability, and compatibility across different platforms. The latest version, Java 17, introduces several language enhancements, performance improvements, and new APIs, further enhancing the capabilities and efficiency of the backend codebase.
4. Lambok: Lambok is employed to reduce boilerplate code and enhance developer productivity. It automatically generates getter and setter methods, reducing the need for manual implementation. Lambok annotations streamline the creation of data models and eliminate the repetitive task of writing repetitive code, resulting in more concise and maintainable code.
5. Spring Data JPA (SQL): Spring Data JPA is utilized for simplified database access and management. It provides an abstraction layer over traditional JDBC (Java Database Connectivity) operations, allowing developers to interact with the database using object-oriented entities and queries. Spring Data JPA automates many common database operations, such as CRUD (Create, Read, Update, Delete), and simplifies complex query construction.
6. Rest Repository Web: The Rest Repository Web module of Spring Boot is employed to expose RESTful APIs for interacting with the backend. It automates the creation of RESTful endpoints based on the defined data models and repositories, reducing manual API development efforts. Rest Repository Web allows clients to perform standard CRUD operations through HTTP protocols, providing a standardized and scalable approach for communication between frontend and backend.
7. MySQL Driver: The MySQL driver is included as a dependency to establish the connection between the backend and the MySQL database. It enables communication and data retrieval from the MySQL database, ensuring efficient and secure data access for the application.

By leveraging Spring Boot, Java 17, Maven, and integrating libraries like Lambok, Spring Data JPA, Rest Repository Web, and MySQL driver, the backend development of the project achieves a robust and efficient infrastructure. Spring Boot simplifies the development and configuration process, while Java provides a powerful and versatile programming language. The additional libraries enhance developer productivity, automate common tasks, and streamline database access, contributing to a scalable and high-performing backend system.

3.1.3. Frontend Development

In the front-end development of my project, Angular was chosen as the primary framework due to its robustness, scalability, and extensive community support. Angular utilizes TypeScript, a statically-typed superset of JavaScript, providing enhanced development capabilities and improved code maintainability.

1. Angular Components: The user interface of my project was built using Angular components. These components encapsulate the HTML, CSS, and TypeScript code required to render specific sections of the application. Each component represents a distinct UI element, such as navigation menus, product listings, blog posts, and user authentication forms.
2. HTML and CSS: HTML (Hypertext Markup Language) is used to structure the content and define the layout of web pages. It is utilized in conjunction with Angular components to create dynamic and responsive user interfaces. CSS (Cascading Style Sheets) is employed to define the visual styles, colors, and layout properties of the project. By leveraging CSS, the appearance of the application can be customized and aligned with the project's design goals.
3. TypeScript: TypeScript is used as the programming language for Angular components. It extends the capabilities of JavaScript by adding static typing, object-oriented features, and improved tooling support. The use of TypeScript allows for better code organization, type checking, and enhanced development productivity.
4. Bootstrap Library: The Bootstrap library is utilized to enhance the visual presentation and responsiveness of the project's user interface. Bootstrap provides a collection of pre-built CSS styles, components, and layout grids, enabling the creation of a consistent and visually appealing frontend. By leveraging Bootstrap's responsive design principles, the application is optimized for various screen sizes and devices.
5. FontAwesome Library: The FontAwesome library is employed to incorporate scalable vector icons into the project. FontAwesome offers a wide range of icons that can be easily customized and styled using CSS classes. These icons are used to enhance the visual representation of various UI elements, providing an intuitive and visually engaging experience for users.

The project's frontend development utilizes Angular, CSS, HTML, TypeScript, Bootstrap, and FontAwesome for a professional, aesthetically pleasing user interface. These libraries enhance visual appeal, responsiveness, and user experience.

3.1.4. Database Management

In my project, MySQL was selected as the database management system (DBMS) to handle data storage and retrieval. The choice of MySQL was based on several considerations, including its reliability, performance, widespread adoption, and compatibility with the project's requirements.

1. MySQL: MySQL is a popular open-source relational database management system. It is widely used in various web applications and offers a comprehensive set of features for efficient data management. MySQL provides robust transactional support, data integrity mechanisms, and excellent scalability, making it suitable for projects of different sizes and complexities.

The decision to use MySQL for the project was driven by its compatibility with the project's requirements. MySQL's relational database model aligns well with the structured nature of

e-commerce and blog-related data. It enables the creation of tables, relationships, and constraints to ensure data consistency and integrity.

3.1.5. Development Tools and Environment

During my project, various development tools and software were utilized to support the development process and enhance productivity. The following tools and software were integral to the project:

1. IDEs and Text Editors:
   a. IntelliJ IDEA (Community Edition 2023.11):

IntelliJ IDEA served as the primary integrated development environment (IDE) for backend development. Its robust features, code analysis capabilities, and seamless integration with Maven and Spring Boot facilitated efficient coding, debugging, and testing of the backend components.

   b. Visual Studio Code:

Visual Studio Code was utilized as the text editor for frontend development. Its lightweight and extensible nature, coupled with a rich ecosystem of extensions, made it a versatile tool for writing and editing HTML, CSS, TypeScript, and other frontend code.

2. Version Control System:
   Git was employed as the version control system to manage the source code. Git provides a distributed and scalable approach to version control, track changes, and merge code seamlessly. Online platforms such as GitHub or GitLab were likely used to host and manage the Git repositories, facilitating code collaboration and version history tracking.

3. Database Management:
   a. MySQL Workbench 8.0 CE:

MySQL Workbench was used as a visual tool for designing, developing, and administering the MySQL database. It provides a graphical interface for schema design, data modeling, query execution, and performance tuning, streamlining the database management tasks.

   b. Postman:

Postman was utilized as an API development and testing tool. It enables the testing and debugging of RESTful APIs, facilitating communication between the frontend and backend components. Postman allows for the sending of HTTP requests, inspecting responses, and managing API collections.

4. Command Line Tools:
   a. Node.js (v16.10.0):

Node.js, specifically its command-line tool, was used to run JavaScript code from the command line. It provides a runtime environment for executing JavaScript outside of a browser, enabling server-side scripting and utility scripts for the project.

b. npm (Node Package Manager) (7.24.0):

npm served as the package manager for the Node.js ecosystem. It allowed for the easy installation, management, and updating of JavaScript libraries, dependencies, and development tools.

c. Tsc (TypeScript Compiler) (Version 4.6.4):

Tsc, the TypeScript compiler, was utilized to compile TypeScript code into JavaScript. It enabled the usage of modern JavaScript features and ensured compatibility with different browsers and environments.

The combination of these development tools and software contributed to a streamlined and efficient development process. IDEs provided powerful coding features and debugging capabilities, while version control systems ensured code collaboration and version management. Database management tools facilitated efficient database design, querying, and administration. Command line tools enhanced scripting capabilities, and npm facilitated easy management of JavaScript libraries and dependencies. Together, these tools supported accelerated development, and helped maintain code quality throughout the project.

3.2. Methodologies Used In Preparing The Thesis

The e-commerce and blog site project thesis involved the use of various methodologies to ensure its successful completion. These methodologies were tailored to fit the unique needs and goals of the project, including product browsing, blog reading, membership functions, and content management. A thorough analysis of the project requirements and extensive research were conducted to gain insights from existing e-commerce and blogging platforms. Using Spring Boot for the backend and Angular for the frontend provided a solid foundation to implement the desired functionality and provide a smooth user experience.

Continuous integration and automated testing were implemented throughout the development process to maintain code quality and ensure stability. MySQL was chosen as the database management system due to its reliability, performance and compatibility with project requirements.

During the development, first database models were created using MySql WorkBench and the related tables were saved in the database. Then the project was started using the visual studio code editor. The front end of the project was created using Angular with HTML, CSS,

Typescript and bootstrap. Afterwards, functions were created using Intellij ide and Spring Boot on the backend and a connection was established with the database and frontend. Finally, the unit and system tests of the project were performed and the errors were corrected.

The combination of these methodologies and appropriate technologies contributed to the successful development and implementation of the e-commerce and blogging site, ensuring efficient project management, iterative development and rigorous testing, ultimately contributing to the completion of a robust and user-friendly platform.

3.3. METHODS USED IN RESEARCHING THE THESIS

In this section, the research methodology employed in this thesis will be elucidated.

3.3.1. Aims and Objectives of the Research

The primary objective of this thesis is to develop a project that combines e-commerce and blog functionalities. The project aims to allow users to browse products and access blog posts without the requirement of membership. To achieve this, various methods and tools were employed throughout the research process, including the analysis of existing blog sites and e-commerce platforms.

3.3.2 Questions of Research

The research was guided by the following key questions:

1. What are the essential tools and methods utilized in the development of a blog site and an e-commerce platform?
2. How can the interface of the website be designed to ensure user-friendliness and effectiveness?

3.3.3 Data Collection and Dataset

During the data collection phase, thorough examination of existing systems was conducted to understand the requirements and needs of the users. Subsequently, a project was designed and implemented accordingly to fulfill these identified needs.To support the development process, five distinct SQL scripts were employed at the project's inception.

These SQL scripts were instrumental in creating datasets for various essential tables within the system. The tables encompassed user information,Post details, product details, product categories, shipment data (order info), as well as country and city data. Each dataset played a crucial role in different aspects of the project's functionality and user experience.

In order to ensure diversity and comprehensive coverage, tertiary sources were utilized to supplement the product dataset and the country-city dataset. This approach enriched the system with a wider range of product offerings and diverse geographical locations.

By leveraging these datasets, the project aimed to provide a comprehensive solution that meets the diverse needs of the users. The datasets were carefully selected and tailored to align with the project's objectives, enabling effective data management and supporting functionalities such as user registration, product browsing, shipment tracking.

The utilization of these datasets, along with the inclusion of tertiary sources to enhance diversity, contributes to the project's ability to deliver a robust and user-centric solution. The relevance and accuracy of the collected data were ensured through meticulous preprocessing steps, ultimately facilitating an enriched user experience and achieving the project's goals.

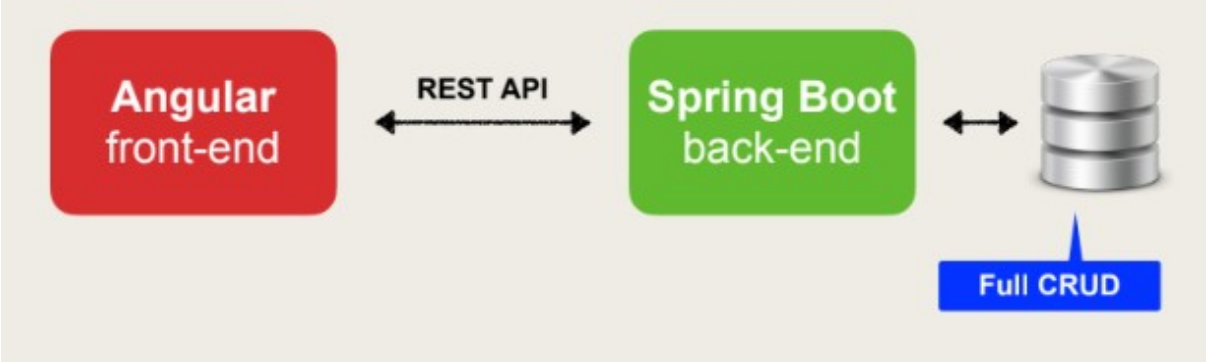
4. Proposed Method

When writing the "Proposed Method" section, you can include the following subheadings to ensure a comprehensive and structured presentation:

4.1. System Overview:

In e-commerce application project with blog support, the MySQL database stores and manages the application's data, the Spring Boot backend handles the business logic and database interactions, and the Angular frontend provides the user interface and communicates with the backend through RESTful APIs, which are exposed by the Spring Boot backend. These APIs define the endpoints and the data exchange format (typically JSON) for communication between the frontend and backend.To establish the connection between them, the Angular application makes HTTP requests to the appropriate API endpoints provided by the Spring Boot backend. For example, when a user wants to view a product or submit an order, the frontend sends a request to the corresponding API endpoint, which then interacts with the MySQL database to fetch or update the relevant data. The backend processes the request, performs the necessary operations on the database, and returns the response to the frontend.Together, these three components work in harmony to deliver a functional and

seamless e-commerce with blog application, enabling users to browse products, make purchases, read blog posts, and perform various other actions.

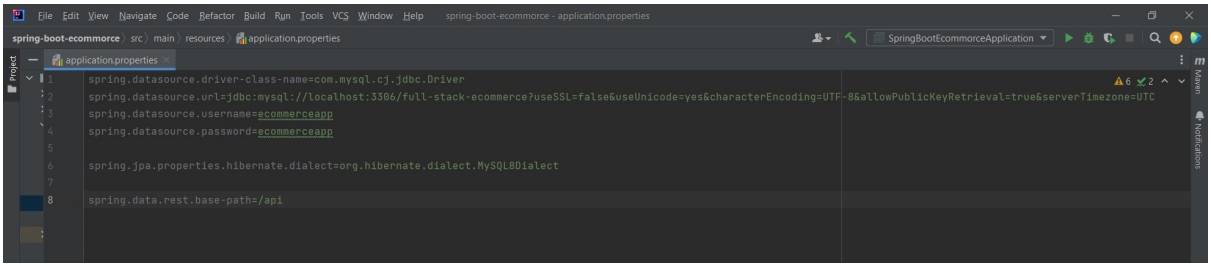


Database structure and interaction:

First, the MySQL database in the project must be added to the Spring Boot project so that it can connect to localhost and interact with the backend. This project was accomplished with the following steps:

1. Connecting MySQL Database to Localhost:
- Downloaded and installed MySQL on the local machine.
- A new database was created by running the `CREATE DATABASE full-stack-ecommerce;` SQL command after starting the MySQL command line client. Here, full-stack-ecommerce represents the database name.
- After the database was created, tables were created and structures were defined using SQL commands.

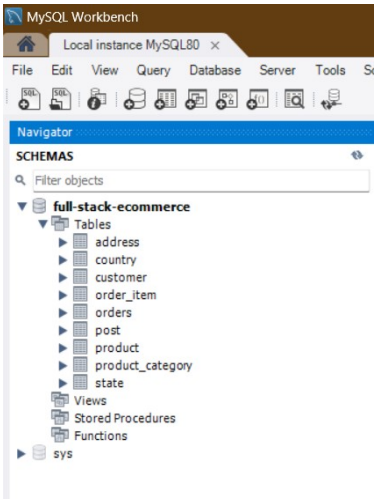
2. Adding MySQL Database to a Spring Boot Project:
- In Spring Boot project, database connection details including URL username, password and driver class name are configured in "../resources/application.properties" file.



- A data access layer was created using Spring Data JPA to interact with the MySQL database, and entity classes representing database tables were used with appropriate annotations to map to database tables (eg "@Entity", "@Table", "@Column").

```
6 usages
@Entity
@Table(name="state")
@Data
public class State {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private int id;
```

After establishing a connection between your Spring Boot application and the MySQL database running on localhost, the necessary database tables for the project were created.



Backend structure and interaction:

contribution of the packages in your Spring Boot project and their interaction with each other, the database, and the frontend:

The project structure in the `com.finalproject.ecommorce` package of the `spring-boot-ecommorce` project contains several essential packages that play a crucial role in the overall functionality and organization of the application.

1. Config Package:
The `config` package typically includes configuration classes that handle various configurations for the application. It contains classes responsible for configuring database connections, security settings, application properties, and other custom configurations.These classes ensure that the application is properly configured and can interact with the necessary resources, such as the database.

2. Controller Package:

The `controller` package contains classes responsible for handling incoming requests and returning appropriate responses to the client. Controllers act as the entry point for different API endpoints or web pages, and they define the application's RESTful. These classes receive requests from the frontend, process the data, invoke the corresponding service methods, and return the results back to the client.

3. DAO Package:
The `dao` (Data Access Object) package houses Interface classes that interact directly with the database. DAO classes typically include methods for performing CRUD (Create, Read, Update, Delete) operations on the database tables.They use technologies like Spring Data JPA to execute SQL queries or leverage object-relational mapping to access and manipulate data.

4. DTO Package:
The `dto` (Data Transfer Object) package contains classes that serve as data containers for transferring data between different layers of the application. DTOs help in encapsulating data and transporting it efficiently without exposing the underlying entities or domain objects.
These classes were often used to define the structure of data sent between the frontend and backend, providing a clear contract for data exchange.

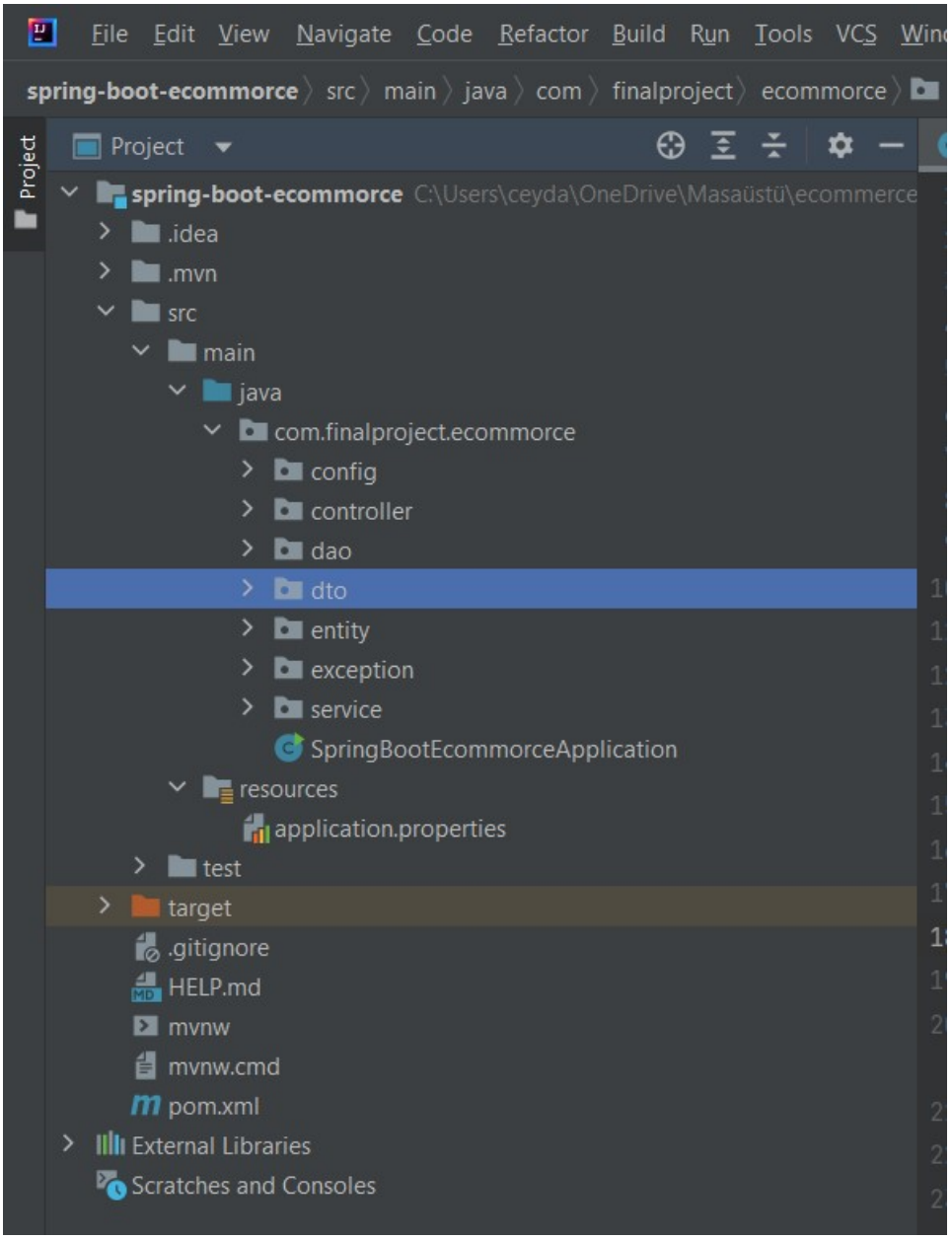
5. Entity Package:
The `entity` package represents the domain model or the data objects that map to database tables.Entity classes were typically annotated with JPA annotations such as `@Entity`, `@Table`, and `@Column`. They define the structure and relationships between different entities and are used for object-relational mapping.

6. Exception Package:
The `exception` package contains custom exception classes that handle specific application-related exceptions or errors.

7. Service Package:
The `service` package houses classes responsible for implementing the business logic of the application. Service classes encapsulate the application's core functionality and are often coordinated with DAO classes to perform data operations.
8. Login Register Package:
The `login register` package focuses on authentication and user management functionalities.It typically includes classes related to user registration, login, password management, and authorization.These classes handle user authentication, session management, and enforce security measures to protect sensitive user information.

Overall, the project structure and the packages within it ensure a modular and organized approach to developing the e-commerce application. They facilitate the interaction between the frontend and backend components by handling requests, processing data, interacting with the database through DAOs, and implementing business logic through services. This

separation of concerns and modular design enable better maintainability, extensibility, and scalability of the application.



Frontend structure and interaction:

The definition of the packages in the frontend part of the project created with Angular and how they contribute to the general functionality, their connections with each other and their structural information are as follows.

1. Common Package:

The `common` package, located in the `../src/app` directory, includes shared resources and common functionality used throughout the application. It typically contains classes that define common data models, constants, utilities, or helper functions used by various components and services. For example, the `Product` class, defined in the `common` folder, represents the data structure for products and their related properties. This package serves as

a centralized place to share common resources and ensures consistency across different parts of the application.

2. Components Package:
The `components` package holds Angular components responsible for rendering specific sections or features of your e-commerce application's user interface. Components are modular building blocks that encapsulate logic and templates to provide a specific functionality or UI element.They include components for displaying products, managing user profiles, handling shopping cart functionality, and more. These components are structured and organized within the package based on their specific purpose and functionality.

3. Services Package:
The `services` package contains files with a `.service.ts` extension that provide specific services and functionality to different parts of the application. Service files encapsulate business logic, interact with the backend APIs, and handle data retrieval, modification, and communication between the frontend and backend. In the project, service files were created for tasks such as retrieving product data, managing user authentication, managing shopping cart operations, and making HTTP requests to the backend.

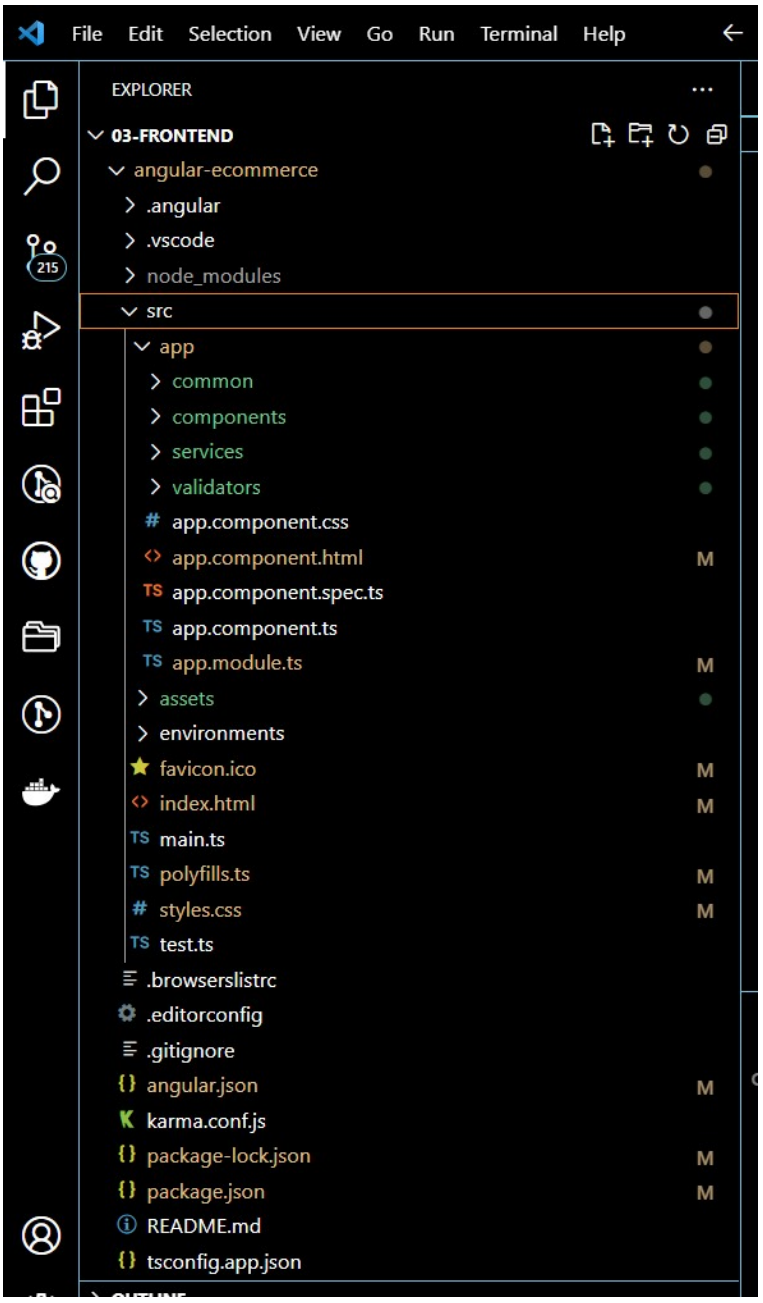
4. Validators Package:
The `validators` package holds exception classes or validation-related files.Validators were used to enforce data validation rules and ensure that the input provided by users meets specific criteria. These classes validate form inputs, handle validation errors, and ensure data integrity. By creating custom validators, Enabled use in the project to apply complex validation rules specific to the application's requirements.

5. Asset Package:
The `asset` package typically contains static files such as images, stylesheets, or other assets used by your Angular application. These files were used to enhance the visual presentation and styling of the application.

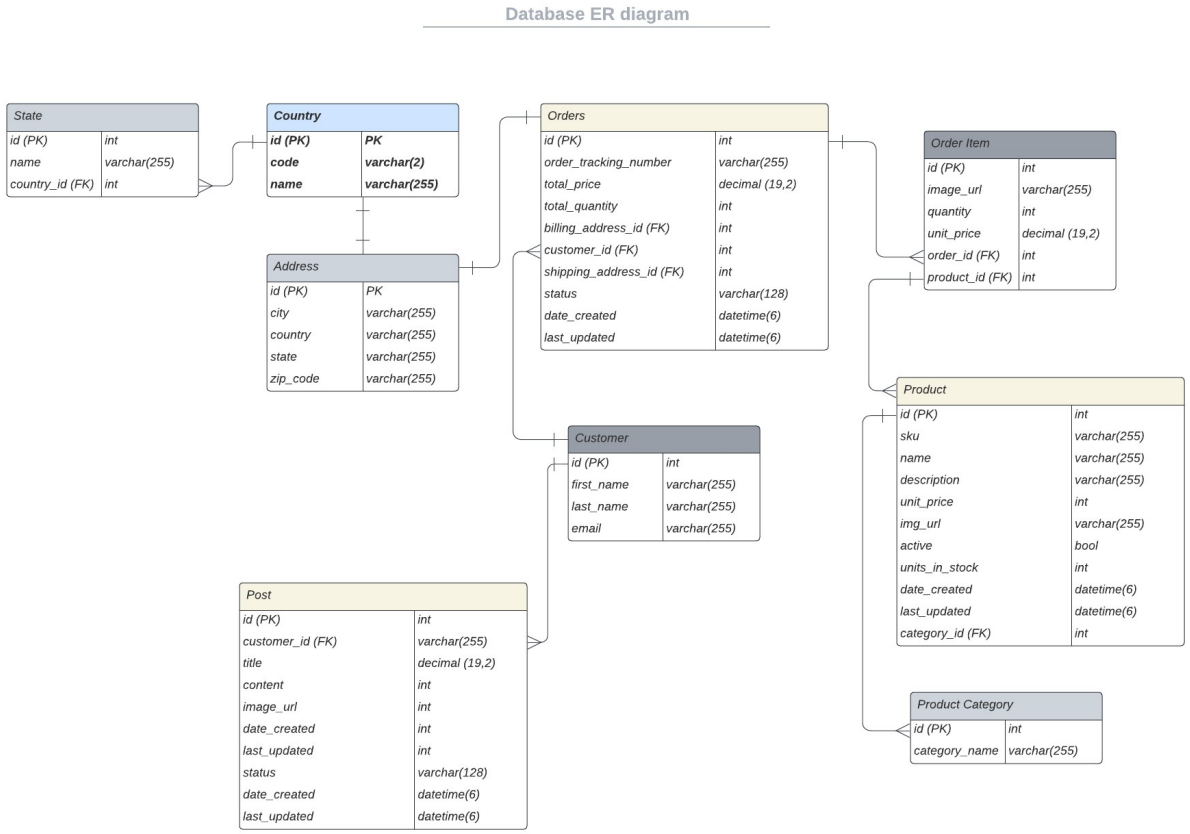
These packages in the Angular project are interconnected to provide a seamless user experience and connect to the backend. Components render the UI and interact with services, which handle data retrieval, manipulation, and communication with the backend APIs. The common package provides shared resources and models, ensuring consistency across the application. Validators help enforce data validation rules to maintain data integrity. The asset package holds static files used for visual enhancement.

To establish a connection with the backend, the service files within the `services` package make HTTP requests to the appropriate backend APIs using Angular's HttpClient module. This allows the frontend to retrieve data, send data, and perform operations on the backend server.

Overall, the organization of these packages within the Angular project ensures modularity, code reusability, and maintainability. It enables efficient collaboration between frontend and backend developers and provides a structured approach to building a functional and connected e-commerce with blog application.



4.2. Database Schema (E-R Diagram):



- The address table has a one-to-many relationship with the country table (country:state).
- The customer table has a one-to-many relationship with the address table (customer:address).
- The orders table has a one-to-many relationship with the address table (billing_address, shipping_address), customer table (customer), and order_item table (order_item:orders).
- The order_item table has a many-to-one relationship with the orders table (order_item:orders) and product table (product:order_item).
- The product table has a many-to-one relationship with the product_category table (category:product).
- The post table has a many-to-one relationship with the customer table (user:customer) for the author information.
-

4.3. User Groups and Roles:

In the application, there are two main user groups or roles: Guests and Members.

1. Guests:
    - Guests refer to users who visit the e-commerce and blog application without logging in or creating an account.
    - As guests, users have limited privileges and access levels within the system.
    - They can browse and view products, read blog posts, and access public information available on the website.
    - However, guests do not have the ability to perform actions such as making purchases, adding posts to the blog, or accessing personalized features.
2. Members:
    - Members are users who have created an account and logged into the application.
    - As members, they enjoy enhanced privileges and access levels compared to guests.
    - Members have the ability to access personalized features such as adding products to their shopping cart, making purchases, writing blog posts, and managing their own posts.

User authentication and authorization are implemented to ensure the security and proper functioning of the system. When a user registers or logs in, their credentials are verified to authenticate their identity. This process helps ensure that only authorized individuals can access the system's functionalities.

Once authenticated, the system grants different levels of access and privileges based on the user's role. Authorization mechanisms are implemented to enforce these access restrictions. For example, members will have access to functionalities like adding products to the shopping cart or creating blog posts, while guests are restricted from performing these actions.

User authentication and authorization are typically managed through a combination of techniques such as username/password validation, session management, and token-based authentication. These mechanisms help safeguard sensitive user information, prevent unauthorized access, and ensure a secure and personalized experience for each user within the e-commerce and blog application.

4. System Workflow:

The system workflow of the application can be summarized as follows, providing a step-by-step explanation from a user's perspective:

1. Accessing the Website:
   - Users initially arrive at the website and are greeted with the e-commerce section.

- The website's menu section displays product categories, allowing users to browse products based on their interests.By default, the first row of products within the selected category is displayed on the screen.

- Users can utilize the search field in the additional header to filter and find specific products.

- Each product is presented as a card, featuring an image, name, and price.

- Under each product card, there is an "Add to Cart" button that enables users to add desired products to their shopping cart.

2. Browsing and Product Details:

- Users can click on the product image or name to access the detailed product page.

- The product details page provides more information about the selected product.

- Similar to the product listings, the product details page also includes an "Add to Cart" button for easy product selection.

- Additionally, a link is available at the bottom of the page, allowing users to return to the shopping page.

3. Blog Section:

- In the footer section of the website, users can find links to various sections, including About Us, Connect Us, Help, and the Blog page.

- Clicking on the "Blog" link directs users to the blog section.

- The blog section showcases blog posts through blog cards, featuring images and titles.

- Users can click on the "Read More" option within each card to access the full details of a particular blog post.

4. User Login and Checkout:

- Upon logging in, users gain access to additional features and functionalities.

- Within the shopping cart, users can proceed to the checkout page by clicking the "Checkout" button.

- The checkout page presents a form for users to fill in their order details.

- After completing the required information, users can finalize their purchase by clicking the "Purchase" button.

5. Blog Interaction:

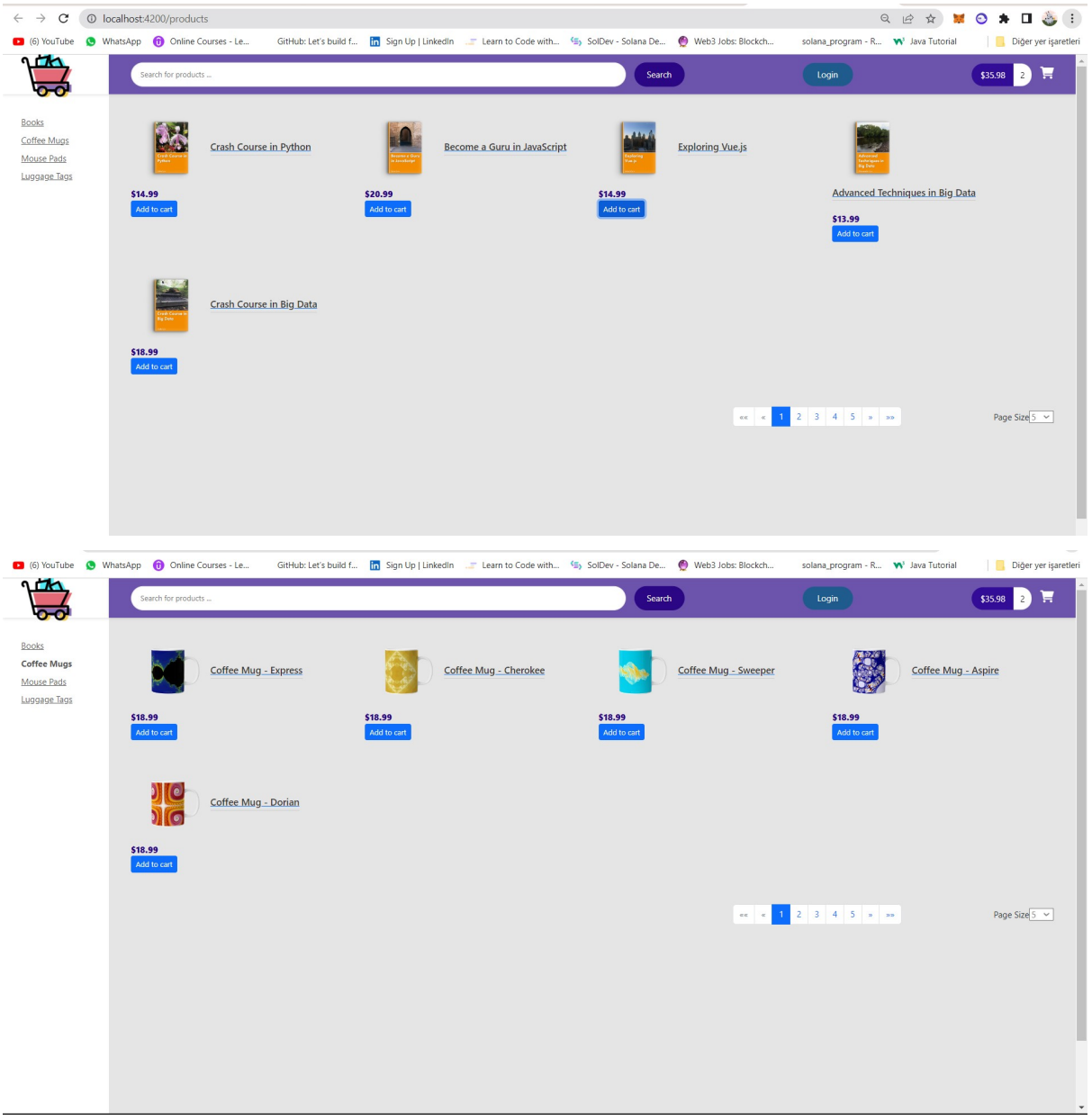
- Within the blog section, users can create their own blog posts by accessing the "Add Post" link in the menu.

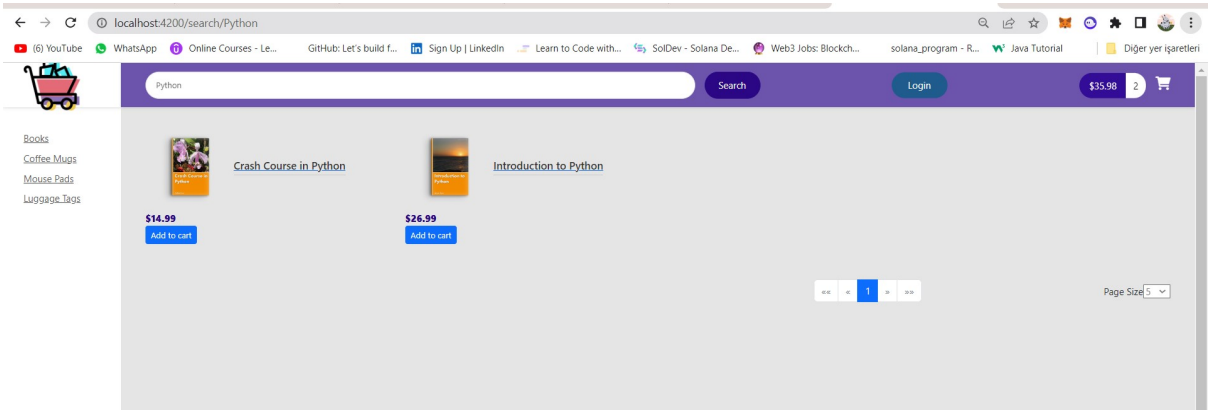
- Users have the ability to manage their blog posts, including deleting or updating the posts they have created.

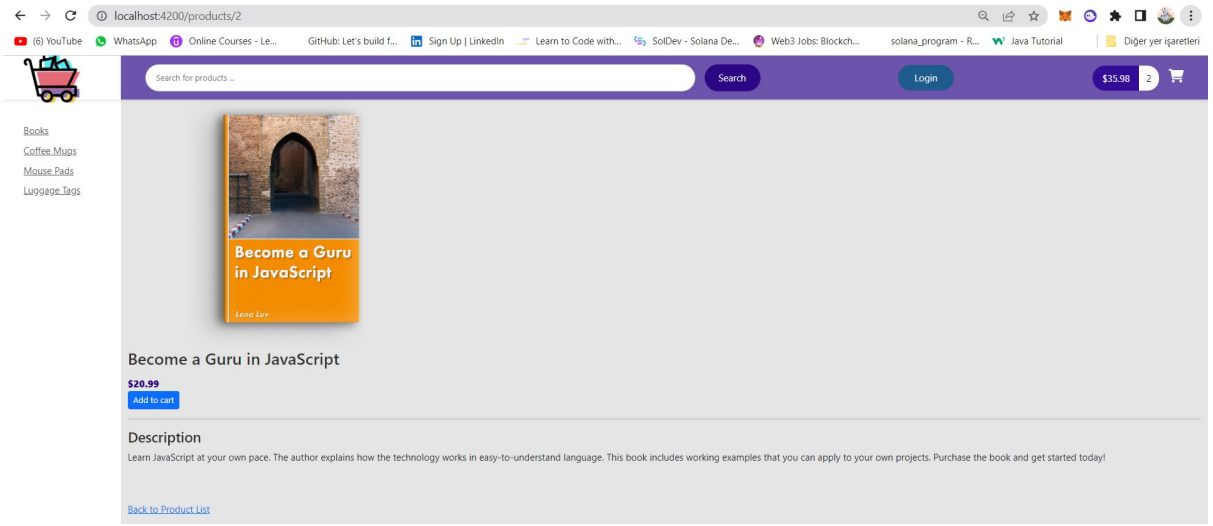
In summary, users begin their journey as guests, exploring the e-commerce section and browsing products. They can add products to their cart, view product details, and filter their search. They can also navigate to the blog section, read blog posts, and access the full post details. Upon logging in, users can proceed to the checkout page for making purchases. Additionally, users can actively participate in the blog section by creating and managing their

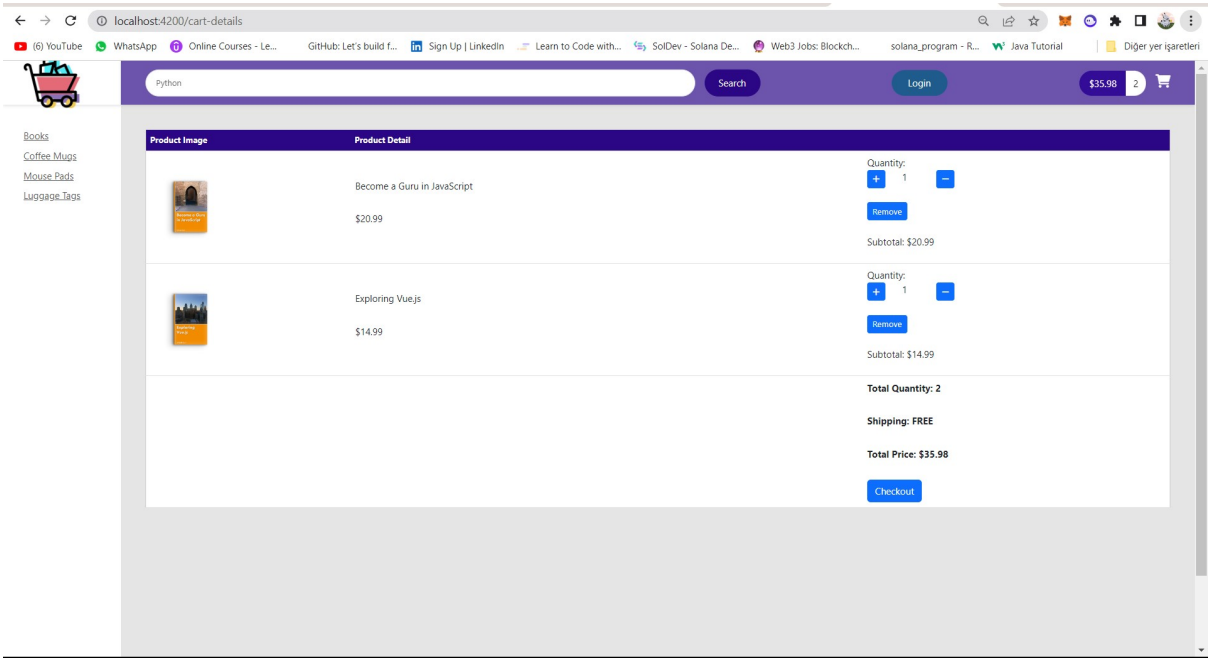
own blog posts. The application seamlessly integrates e-commerce and blog functionalities, providing a comprehensive user experience.

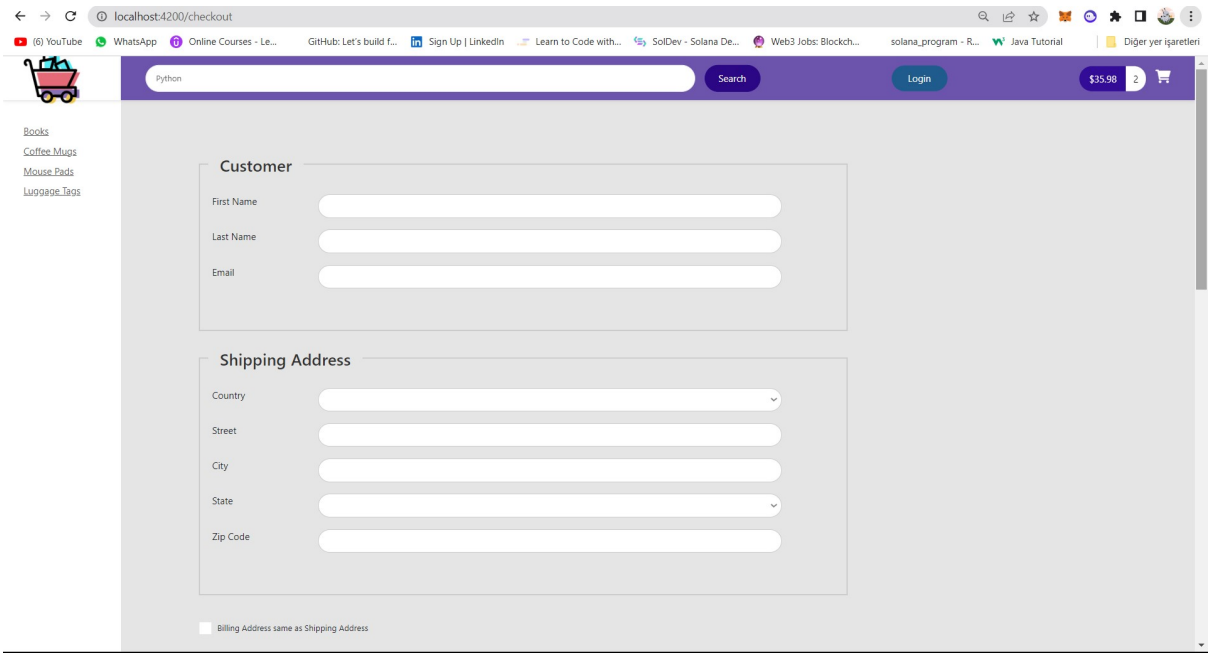
4.5. Screen Outputs

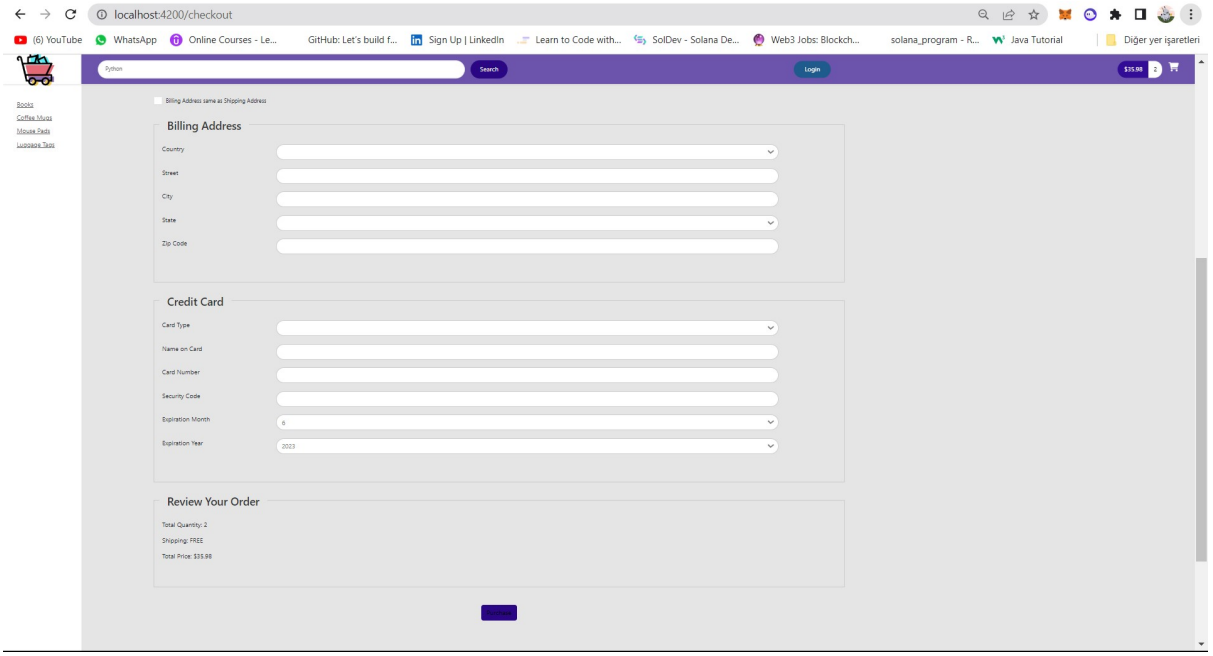


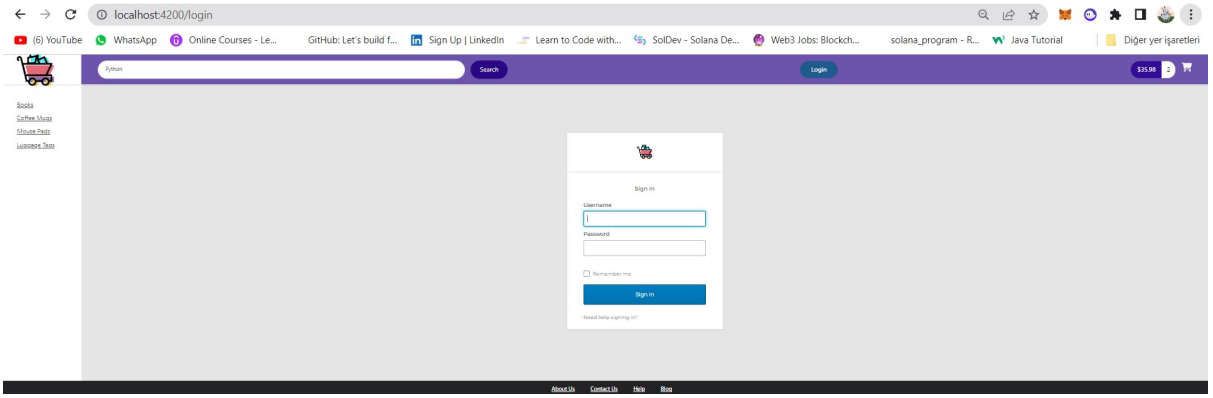

localhost:4200/search/Python
Python
Search
Login
$35.98
2
Books
Coffee Mugs
Mouse Pads
Luggage Tags
Crash Course in Python
$14.99
Add to cart
Introduction to Python
$26.99
Add to cart
Page Size

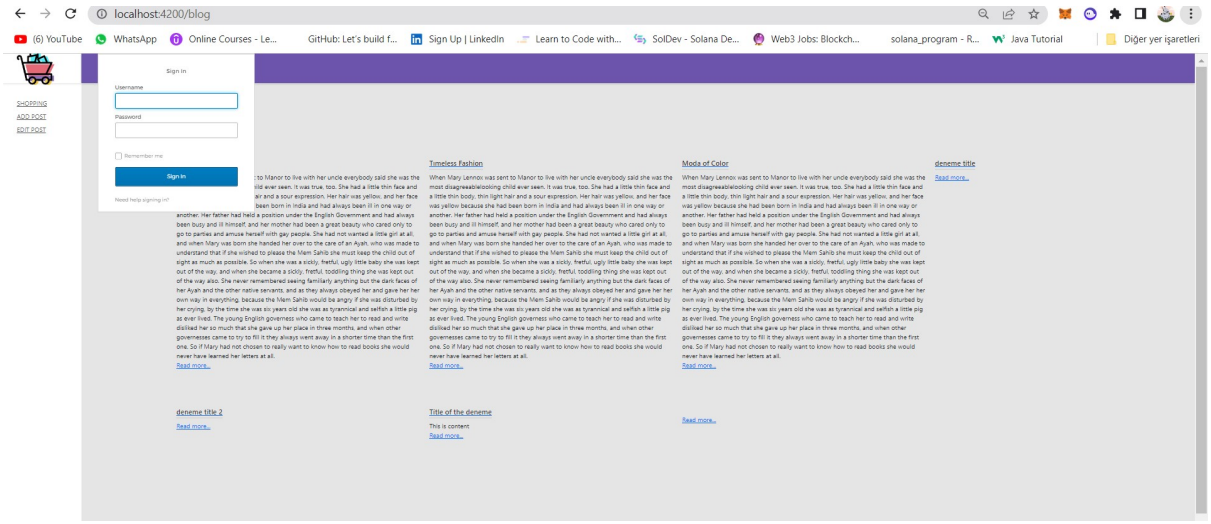

localhost:4200/products/2
Search for products ...
Search
Login
$35.98
2
Books
Coffee Mugs
Mouse Pads
Luggage Tags
Become a Guru in JavaScript
Become a Guru in JavaScript
$20.99
Add to cart
Description
Learn JavaScript at your own pace. The author explains how the technology works in easy-to-understand language. This book includes working examples that you can apply to your own projects. Purchase the book and get started today!
Back to Product List

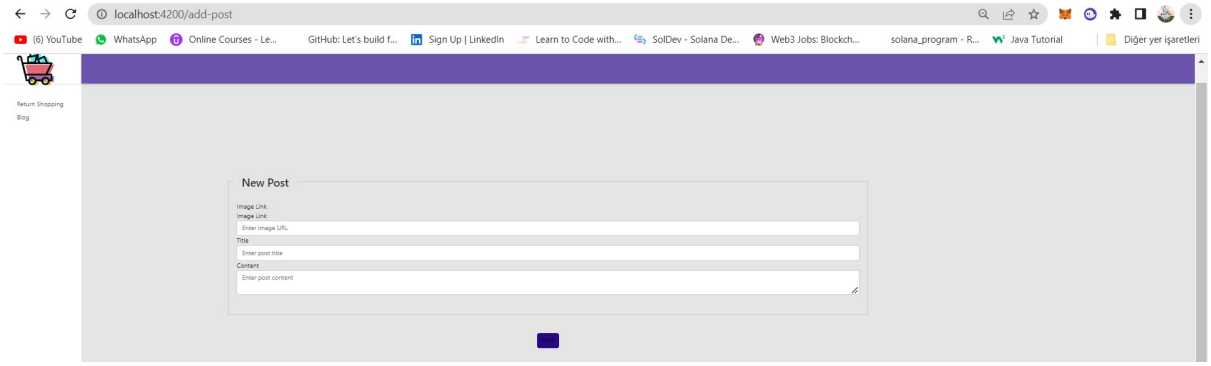

localhost:4200/cart-details
Python
Search
Login
$35.98
2
Books
Coffee Mugs
Mouse Pads
Luggage Tags
Product Image
Product Detail
Become a Guru in JavaScript
$20.99
Quantity:
1
Remove
Subtotal: $20.99
Exploring Vue.js
$14.99
Quantity:
1
Remove
Subtotal: $14.99
Total Quantity: 2
Shipping: FREE
Total Price: $35.98
Checkout

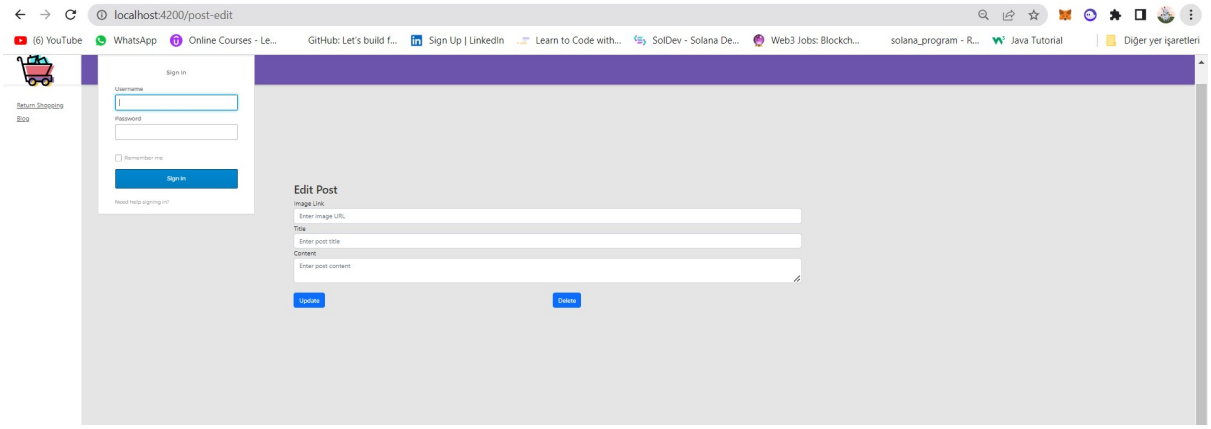

localhost:4200/checkout
Python
Search
Login
$35.98
2
Books
Coffee Mugs
Mouse Pads
Luggage Tags
Customer
First Name
Last Name
Email
Shipping Address
Country
Street
City
State
Zip Code
Billing Address same as Shipping Address


localhost:4200/checkout
Billing Address
Credit Card
Review Your Order


localhost:4200/login
Sign in
Username
Password
Remember me
Sign in
About Us
Contact Us
Help
Blog







4.6 Code explanations.

1.E-commerce side

Product operations

In the application, necessary functions for backend and frontend are created by using the product and product_category tables in the database as entity classes in order to show the list of products to the user depending on the category, to search for products and to see product details.

First of all, ProductRepository and ProductCategoryRepository interfaces were created as follows in order to perform database operations with the backend.

İn ProductRepository.java ;

```
@CrossOrigin("http://localhost:4200")
public interface ProductRepository extends JpaRepository<Product, Long> {

Page<Product> findByCategoryId(@Param("id") Long id, Pageable pageable);

Page<Product> findByNameContaining(@Param("name") String name, Pageable pageable);

}
```

The `ProductRepository` interface is a part of the Spring Data JPA framework and is used to define methods for interacting with the `Product` entity in the database. In this specific code snippet, there are two methods defined:

1. `findByCategoryId`: This method is used to retrieve a page of products based on a specific category ID. It accepts the category ID as a parameter (`id`) and a `Pageable` object that specifies the page number, page size, sorting, etc. It performs a query to fetch products that belong to the specified category ID and returns the result as a `Page` object.

2. `findByNameContaining`: This method is used to retrieve a page of products based on a partial match of the product name. It accepts the partial name as a parameter (`name`) and a `Pageable` object. It performs a query to search for products whose names contain the specified partial name and returns the result as a `Page` object.

These methods leverage the Spring Data JPA framework's automatic query generation based on method names. By following the naming convention, you can easily define queries without writing explicit SQL statements. The `JpaRepository` interface provides the necessary functionality for executing these queries and performing CRUD operations on the `Product` entity.

In ProductCategoryRepository.java

```
@CrossOrigin("http://localhost:4200")
@RepositoryRestResource(collectionResourceRel = "productCategory", path =
"product-category")
```

```
public interface ProductCategoryRepository extends JpaRepository<ProductCategory, Long>
{
}
```

The ProductCategoryRepository interface plays a crucial role in handling product categories in a backend application. By using annotations such as @CrossOrigin and @RepositoryRestResource, it enables the repository to provide RESTful endpoints for performing CRUD operations on product categories. These endpoints allow the frontend application to interact with the backend and manage product categories through standard HTTP methods like GET, POST, PUT, and DELETE. Essentially, the repository acts as a bridge between the frontend and backend, allowing easy access and manipulation of product category data.

In the Front End, components have been created in order to connect to these methods and reflect them to the users. The ones related to listing products are Product-List, Product-Details, Product-Category-Menu and Search packages. These packages contain .ts files that communicate with the service and .html files that are reflected on the user side. The first Product.service.ts file was created in order to connect with these files and the backend in Product operations.

In Product.service.ts

```
getProduct(theProductId: number): Observable<Product> {

  // need to build URL based on product id
  const productUrl = `${this.baseUrl}/${theProductId}`;

  return this.httpClient.get<Product>(productUrl);
}
```

The `getProduct` method in the `Product.service.ts` file is responsible for retrieving a specific product based on its ID. It takes the product ID as a parameter.

Inside the method, a URL is constructed using the base URL of the API and the provided product ID. Then, an HTTP GET request is made to that URL using the Angular `httpClient` module. The response from the server is expected to be of type `Product`, and it is returned as an observable.

In summary, this method allows you to fetch a single product from the server by making an HTTP GET request with the product ID as a parameter. The response is returned as an observable, which can be subscribed to in order to retrieve the product data.

```
getProductList(theCategoryId: number): Observable<Product[]> {
```

```
  // need to build URL based on category id
  const searchUrl = `${this.baseUrl}/search/findByCategoryId?id=${theCategoryId}`;

  return this.getProducts(searchUrl);
}
```

The `getProductList` method in the `Product.service.ts` file is used to retrieve a list of products based on a specific category ID. It takes the category ID as a parameter.

Inside the method, a URL is constructed using the base URL of the API and the provided category ID. The URL is formatted in a way that corresponds to the API endpoint responsible for fetching products by category ID.

Then, the `getProducts` method is called passing the constructed URL as a parameter. This method is responsible for making an HTTP GET request to the server using the `httpClient` module and returning the response as an observable of `Product[]` (an array of products).

In summary, this method enables you to fetch a list of products from the server based on a specific category ID. It constructs the appropriate URL for the API endpoint and utilizes the `getProducts` method to make the HTTP request and return the response as an observable.

```
searchProducts(theKeyword: string): Observable<Product[]> {

  // need to build URL based on the keyword
  const searchUrl =
`${this.baseUrl}/search/findByNameContaining?name=${theKeyword}`;

  return this.getProducts(searchUrl);
}
```

The `searchProducts` method in the `Product.service.ts` file is used to search for products based on a keyword. It takes the keyword as a parameter.

Inside the method, a URL is constructed using the base URL of the API and the provided keyword. The URL is formatted in a way that corresponds to the API endpoint responsible for searching products by name containing the keyword.

Then, the `getProducts` method is called passing the constructed URL as a parameter. This method is responsible for making an HTTP GET request to the server using the `httpClient` module and returning the response as an observable of `Product[]` (an array of products).

In summary, this method allows you to search for products based on a keyword. It constructs the appropriate URL for the API endpoint and utilizes the `getProducts` method to make the HTTP request and return the response as an observable.

```
getProductListPaginate(thePage: number,
                       thePageSize: number,
                       theCategoryId: number): Observable<GetResponseProducts> {

  // need to build URL based on category id, page and size
  const searchUrl = `${this.baseUrl}/search/findByCategoryId?id=${theCategoryId}`
                  + `&page=${thePage}&size=${thePageSize}`;

  return this.httpClient.get<GetResponseProducts>(searchUrl);
}
```

The `getProductListPaginate` method in the `Product.service.ts` file is used to retrieve a paginated list of products based on a category ID, page number, and page size. It takes these parameters as inputs.

Inside the method, a URL is constructed using the base URL of the API and the provided category ID, page number, and page size. The URL is formatted in a way that corresponds to the API endpoint responsible for retrieving products by category ID with pagination.

Then, the `httpClient` module is used to make an HTTP GET request to the server using the constructed URL. The response is expected to be of type `GetResponseProducts`, which represents the response containing the paginated list of products.

In summary, this method allows you to retrieve a paginated list of products based on a category ID. It constructs the appropriate URL for the API endpoint with pagination parameters and makes an HTTP GET request to the server. The response, containing the paginated list of products, is returned as an observable of type `GetResponseProducts`.

```
searchProductsPaginate(thePage: number,
                       thePageSize: number,
                       theKeyword: string): Observable<GetResponseProducts> {
```

```
  // need to build URL based on keyword, page and size
  const searchUrl = `${this.baseUrl}/search/findByNameContaining?name=${theKeyword}`
                  + `&page=${thePage}&size=${thePageSize}`;

  return this.httpClient.get<GetResponseProducts>(searchUrl);
}
```

The `searchProductsPaginate` method is responsible for performing a paginated search for products based on a given keyword. It takes in three parameters: `thePage`, `thePageSize`, and `theKeyword`.

Inside the method, a URL is constructed using the base URL (`this.baseUrl`) and the provided parameters. The URL is built in a way that it incorporates the keyword, page number, and page size as query parameters.

Then, an HTTP GET request is made to the constructed search URL using the `httpClient` from Angular's `HttpClient` module. The response from the server is expected to be of type `GetResponseProducts`, which represents the response containing paginated product data.

The method returns an `Observable` of type `GetResponseProducts`, allowing the caller to subscribe to the result and receive the response asynchronously.

Overall, the `searchProductsPaginate` method facilitates the search functionality by constructing the appropriate URL and making an HTTP request to retrieve paginated product data based on the provided keyword.

```
private getProducts(searchUrl: string): Observable<Product[]> {
  return this.httpClient.get<GetResponseProducts>(searchUrl).pipe(map(response =>
response._embedded.products));
 }

 getProductCategories(): Observable<ProductCategory[]> {

  return this.httpClient.get<GetResponseProductCategory>(this.categoryUrl).pipe(
    map(response => response._embedded.productCategory)
  );
 }
```

The `getProducts` method is a private helper method used to retrieve a list of products based on a provided search URL. It takes in the `searchUrl` as a parameter, which represents the URL to fetch the products from.

Inside the method, an HTTP GET request is made to the `searchUrl` using the `httpClient` from Angular's `HttpClient` module. The response from the server is expected to be of type `GetResponseProducts`, which contains the paginated product data.

The method uses the `map` operator from the RxJS library to transform the response into an array of products (`Product[]`). It extracts the products from the response's `_embedded.products` property.

The `getProducts` method returns an `Observable` of type `Product[]`, allowing the caller to subscribe to the result and receive the array of products asynchronously.

The `getProductCategories` method is responsible for fetching the list of product categories. It makes an HTTP GET request to the `categoryUrl` using the `httpClient`. The response from the server is expected to be of type `GetResponseProductCategory`, which represents the response containing the product categories.

Similar to the `getProducts` method, the `map` operator is used to transform the response into an array of product categories (`ProductCategory[]`). It extracts the product categories from the response's `_embedded.productCategory` property.

The `getProductCategories` method returns an `Observable` of type `ProductCategory[]`, allowing the caller to subscribe to the result and receive the array of product categories asynchronously.

These methods provide the necessary functionality to retrieve products and product categories from the server, allowing the frontend to display and work with the data effectively.

After the basic service functions for product listing were created in this way, these functions were subscribed to with .ts files in the relevant components and filtered data was obtained. For example;

```
ngOnInit() {
  this.route.paramMap.subscribe(() => {
    this.listProducts();
  });
}

listProducts() {
```

```
  this.searchMode = this.route.snapshot.paramMap.has('keyword');

  if (this.searchMode) {
    this.handleSearchProducts();
  }
  else {
    this.handleListProducts();
  }

}
```

In the `ngOnInit` method of the component, a subscription is created to the `paramMap` observable of the `route` object. This allows the component to react to changes in the route parameters.

Inside the `listProducts` method, the `searchMode` flag is set based on whether the 'keyword' parameter is present in the current route. If the 'keyword' parameter is present, it means the user is performing a search for products.

If the `searchMode` is true, the `handleSearchProducts` method is called to handle the search operation. This method is responsible for retrieving the search keyword from the route parameters and fetching the matching products.

If the `searchMode` is false, it means the user is not performing a search but rather listing all products. In this case, the `handleListProducts` method is called. This method is responsible for fetching and displaying all the products.

By using the `paramMap` observable and checking the presence of the 'keyword' parameter, the component dynamically determines whether to perform a search or list all products based on the current route. This allows the component to handle both scenarios without duplicating code or creating separate routes.

Shopping Cart Functions

In the project, shopping card functions were created to see product details such as adding products to the cart, deleting products, and the total amount, and to be directed to the checkout page if necessary. For these functions, Cart Status and Cart Detail components were created in the frontend and Cart.Service.Ts file was created for their embedded functions.

In Cart.Service.Ts

```
addToCart(theCartItem: CartItem) {

  let alreadyExistsInCart: boolean = false;
  let existingCartItem: CartItem = new CartItem();

  if (this.cartItems.length > 0) {

    existingCartItem = this.cartItems.find( tempCartItem => tempCartItem.id ===
theCartItem.id )!;

    alreadyExistsInCart = (existingCartItem != undefined);
  }

  if (alreadyExistsInCart) {
    existingCartItem.quantity++;
  }
  else {
    this.cartItems.push(theCartItem);
  }

  this.computeCartTotals();
 }
```

The `addToCart` method in the `CartService` is responsible for adding a new `CartItem` to the shopping cart. Here's a breakdown of how it works:

1. It takes the `theCartItem` parameter, which represents the item to be added to the cart.
2. It initializes a flag `alreadyExistsInCart` to `false` and creates an empty `existingCartItem`.
3. It checks if the `cartItems` array already contains items.
4. If there are existing items in the cart, it searches for an item with the same `id` as `theCartItem` using the `find` method on the `cartItems` array. If a matching item is found, it assigns it to `existingCartItem` and sets `alreadyExistsInCart` to `true`.
5. If `alreadyExistsInCart` is `true`, it means the item already exists in the cart. In this case, it increments the `quantity` property of `existingCartItem`.
6. If `alreadyExistsInCart` is `false`, it means the item is not in the cart yet. In this case, it adds `theCartItem` to the `cartItems` array.
7. Finally, it calls the `computeCartTotals` method, which calculates the total quantity and price of all items in the cart.

In summary, the `addToCart` method checks if an item already exists in the cart based on its `id`. If the item is already in the cart, it increments the quantity. If not, it adds the item to the cart. After modifying the cart items, it updates the cart totals.

```
computeCartTotals() {

  let totalPriceValue: number = 0;
  let totalQuantityValue: number = 0;

  for (let currentCartItem of this.cartItems) {
    totalPriceValue += currentCartItem.quantity * currentCartItem.unitPrice!;
    totalQuantityValue += currentCartItem.quantity;
  }

  this.totalPrice.next(totalPriceValue);
  this.totalQuantity.next(totalQuantityValue);

  this.logCartData(totalPriceValue, totalQuantityValue);
 }
```

The `computeCartTotals` method in the `CartService` is responsible for calculating the total price and total quantity of all items in the shopping cart. Here's how it works:

1. It initializes variables `totalPriceValue` and `totalQuantityValue` to 0, which will store the calculated values.
2. It iterates over each `currentCartItem` in the `cartItems` array.
3. For each `currentCartItem`, it multiplies the quantity (`currentCartItem.quantity`) by the unit price (`currentCartItem.unitPrice`) and adds it to the `totalPriceValue`.
4. It also increments the `totalQuantityValue` by the quantity of the `currentCartItem`.
5. After calculating the totals, it updates the `totalPrice` and `totalQuantity` BehaviorSubjects with the new values using the `next` method. This allows other components or services subscribed to these BehaviorSubjects to receive the updated values.
6. It calls the `logCartData` method, which logs the updated cart totals for debugging or logging purposes.

In summary, the `computeCartTotals` method iterates over the cart items, calculates the total price and quantity, updates the corresponding BehaviorSubjects, and logs the cart data.

```
logCartData(totalPriceValue: number, totalQuantityValue: number) {

  console.log('Contents of the cart');
```

```
    for (let tempCartItem of this.cartItems) {
      const subTotalPrice = tempCartItem.quantity * tempCartItem.unitPrice!;
      console.log(`name: ${tempCartItem.name}, quantity=${tempCartItem.quantity},
unitPrice=${tempCartItem.unitPrice}, subTotalPrice=${subTotalPrice}`);
    }

    console.log(`totalPrice: ${totalPriceValue.toFixed(2)}, totalQuantity:
${totalQuantityValue}`);
    console.log('----');
  }
```

The `logCartData` method in the `CartService` is responsible for logging the contents of the shopping cart, including each cart item's details, as well as the total price and total quantity. Here's how it works:

1. It starts by printing the header message "Contents of the cart" to indicate the beginning of the cart data log.
2. It then iterates over each `tempCartItem` in the `cartItems` array.
3. For each `tempCartItem`, it calculates the subtotal price by multiplying the quantity (`tempCartItem.quantity`) by the unit price (`tempCartItem.unitPrice`).
4. It logs the name, quantity, unit price, and subtotal price of the `tempCartItem` using the `console.log` function.
5. After logging all the cart items, it logs the total price and total quantity, which are passed as parameters to the method.
6. Finally, it prints a line of dashes as a separator to visually distinguish between different log outputs.

In summary, the `logCartData` method logs the contents of the shopping cart, including each item's details, as well as the total price and total quantity, providing a snapshot of the cart's current state for debugging or logging purposes.

```
decrementQuantity(theCartItem: CartItem) {

    theCartItem.quantity--;

    if (theCartItem.quantity === 0) {
      this.remove(theCartItem);
    }
    else {
      this.computeCartTotals();
    }
```

```
}
```

The `decrementQuantity` method in the `CartService` is responsible for decrementing the quantity of a given cart item. Here's how it works:

1. It takes a `theCartItem` parameter, which represents the cart item whose quantity needs to be decremented.
2. It decreases the quantity of `theCartItem` by one using the decrement operator (`theCartItem.quantity--`).
3. It checks if the quantity of `theCartItem` has reached zero.
4. If the quantity becomes zero, it calls the `remove` method to remove the cart item from the cart since there are no more items of that type.
5. If the quantity is still greater than zero, it calls the `computeCartTotals` method to recalculate the total price and total quantity of the cart, reflecting the updated quantity of `theCartItem`.

In summary, the `decrementQuantity` method allows for decrementing the quantity of a cart item. If the quantity reaches zero, the item is removed from the cart, otherwise, the cart totals are recalculated to reflect the updated quantity.

```
remove(theCartItem: CartItem) {

  // get index of item in the array
  const itemIndex = this.cartItems.findIndex( tempCartItem => tempCartItem.id ===
theCartItem.id );

  // if found, remove the item from the array at the given index
  if (itemIndex > -1) {
    this.cartItems.splice(itemIndex, 1);

    this.computeCartTotals();
  }
 }
```

The `remove` method in the `CartService` is responsible for removing a cart item from the cart. Here's how it works:

1. It takes a `theCartItem` parameter, which represents the cart item to be removed.
2. It uses the `findIndex` method to find the index of the cart item in the `cartItems` array based on its `id`.

3. If the cart item is found (i.e., `itemIndex` is greater than -1), it uses the `splice` method to remove the item from the `cartItems` array at the given index.
4. After removing the item, it calls the `computeCartTotals` method to recalculate the total price and total quantity of the cart.

In summary, the `remove` method removes a specific cart item from the cart by finding its index in the `cartItems` array and then removing it using the `splice` method. The cart totals are then recomputed to reflect the updated cart after the removal.

Example for components to subscribe to Cart.service;

```
ngOnInit(): void {
  this.listCartDetails();
 }

 listCartDetails() {

  // get a handle to the cart items
  this.cartItems = this.cartService.cartItems;

  // subscribe to the cart totalPrice
  this.cartService.totalPrice.subscribe(
    data => this.totalPrice = data
  );

  // subscribe to the cart totalQuantity
  this.cartService.totalQuantity.subscribe(
    data => this.totalQuantity = data
  );

  // compute cart total price and quantity
  this.cartService.computeCartTotals();
 }

 incrementQuantity(theCartItem: CartItem) {
  this.cartService.addToCart(theCartItem);
 }

 decrementQuantity(theCartItem: CartItem) {
  this.cartService.decrementQuantity(theCartItem);
```

```
}

 remove(theCartItem: CartItem) {
   this.cartService.remove(theCartItem);
 }
```

Order creation and purchases

Address, country and states, order, order_item and costumer entity classes created based on the tables in the database were used to create the order. Since this process requires a database - backend connection, first CheckOut controls, State , Country Customer repository, Purchase and PurchaseResponse Dtos and finally Checkout service files were created in spring boot.

In Checkout service

```
@Override
@Transactional
public PurchaseResponse placeOrder(Purchase purchase) {

// retrieve the order info from dto
Order order = purchase.getOrder();

// generate tracking number
String orderTrackingNumber = generateOrderTrackingNumber();
order.setOrderTrackingNumber(orderTrackingNumber);

// populate order with orderItems
Set<OrderItem> orderItems = purchase.getOrderItems();
orderItems.forEach(item -> order.add(item));

// populate order with billingAddress and shippingAddress
order.setBillingAddress(purchase.getBillingAddress());
order.setShippingAddress(purchase.getShippingAddress());

// populate customer with order
Customer customer = purchase.getCustomer();
customer.add(order);

// save to the database
customerRepository.save(customer);

// return a response
return new PurchaseResponse(orderTrackingNumber);
}
```

```
private String generateOrderTrackingNumber() {

// generate a random UUID number (UUID version-4)
// For details see: https://en.wikipedia.org/wiki/Universally_unique_identifier
//
return UUID.randomUUID().toString();
}
```

The `placeOrder` method is responsible for placing an order based on the information provided in the `Purchase` object. Here's how it works:

1. It retrieves the `Order` object from the `purchase` DTO.
2. It generates a unique order tracking number using the `generateOrderTrackingNumber` method.
3. It sets the order tracking number to the `Order` object.
4. It retrieves the order items from the `purchase` DTO and adds them to the `Order` object.
5. It populates the billing address and shipping address of the `Order` object from the `purchase` DTO.
6. It retrieves the customer from the `purchase` DTO and adds the order to the customer's order list.
7. It saves the customer to the database using the `customerRepository`.
8. Finally, it returns a `PurchaseResponse` object containing the order tracking number.

The `generateOrderTrackingNumber` method generates a random UUID (Universally Unique Identifier) version 4, which serves as a unique identifier for the order.

In summary, the `placeOrder` method processes the purchase information, associates it with a customer, generates an order tracking number, and saves the order to the database. It then returns the order tracking number as a response.

```
private CheckoutService checkoutService;

public CheckoutController(CheckoutService checkoutService) {
this.checkoutService = checkoutService;
}

@PostMapping("/purchase")
public PurchaseResponse placeOrder(@RequestBody Purchase purchase) {

PurchaseResponse purchaseResponse = checkoutService.placeOrder(purchase);

return purchaseResponse;
}
```

The `CheckoutController` class is responsible for handling the HTTP requests related to the checkout process. Let's break down the code:

1. The `CheckoutService` is injected into the controller via the constructor.
2. The `placeOrder` method is annotated with `@PostMapping("/purchase")`, indicating that it handles POST requests to the "/purchase" endpoint.
3. The `@RequestBody` annotation is used to bind the request body to the `Purchase` object, which contains the necessary information for placing an order.
4. Inside the `placeOrder` method, the `checkoutService.placeOrder(purchase)` method is called to process the purchase and retrieve a `PurchaseResponse` object.
5. The `purchaseResponse` is returned as the response to the client.

In summary, when a POST request is made to the "/purchase" endpoint with the purchase information in the request body, the `placeOrder` method in the `CheckoutController` is invoked. It delegates the processing of the purchase to the `checkoutService` and returns the resulting `PurchaseResponse`.

On the Frontend, the Checkout Service class was created to interact with the Form Service and the above functions to receive information from the Form file.

In Form Service

```
private countriesUrl = 'http://localhost:8080/api/countries';
 private statesUrl = 'http://localhost:8080/api/states';

 constructor(private httpClient: HttpClient) { }

 getCountries(): Observable<Country[]> {

  return this.httpClient.get<GetResponseCountries>(this.countriesUrl).pipe(
   map(response => response._embedded.countries)
  );
 }

 getStates(theCountryCode: string): Observable<State[]> {

  // search url
  const searchStatesUrl =
`${this.statesUrl}/search/findByCountryCode?code=${theCountryCode}`;
```

```
  return this.httpClient.get<GetResponseStates>(searchStatesUrl).pipe(
    map(response => response._embedded.states)
  );
}


getCreditCardMonths(startMonth: number): Observable<number[]> {

  let data: number[] = [];

  // build an array for "Month" dropdown list
  // - start at current month and loop until

  for (let theMonth = startMonth; theMonth <= 12; theMonth++) {
    data.push(theMonth);
  }

  return of(data);
}

getCreditCardYears(): Observable<number[]> {

  let data: number[] = [];

  // build an array for "Year" downlist list
  // - start at current year and loop for next 10 years

  const startYear: number = new Date().getFullYear();
  const endYear: number = startYear + 10;

  for (let theYear = startYear; theYear <= endYear; theYear++) {
    data.push(theYear);
  }

  return of(data);
}
```

```
}

interface GetResponseCountries {
  _embedded: {
    countries: Country[];
  }
}

interface GetResponseStates {
  _embedded: {
    states: State[];
  }
}
```

In CheckOut Service

```
export class CheckoutService {

  private purchaseUrl = 'http://localhost:8080/api/checkout/purchase';

  constructor(private httpClient: HttpClient) { }

  placeOrder(purchase: Purchase): Observable<any> {
    return this.httpClient.post<Purchase>(this.purchaseUrl, purchase);
```

The `placeOrder` method in the frontend service is responsible for sending a POST request to the backend server to place an order. Let's break down the code:

1. The method accepts a `purchase` object of type `Purchase`, which contains the necessary information for placing an order.
2. The `httpClient.post` method is used to send a POST request to the specified `purchaseUrl`, which represents the endpoint for placing an order on the backend server.
3. The `purchase` object is included as the body of the request.
4. The return type of the method is an `Observable` that emits the response from the server. The response type is specified as `any`, indicating that it can be of any type.

In summary, when the `placeOrder` method is called, it sends a POST request to the backend server with the `purchase` object as the request body. The method returns an `Observable` that can be subscribed to in order to receive the response from the server.

Blog Side

Listing the posts in the blog section and seeing the post details work with the same logic as the product functions we have created. For Product Add and product edit functions
Post service Post Controller Post repository classes have been created in the backend.

In Post Service

```
@Autowired
private PostRepository postRepository;

@Transactional
public void createPost(PostDto postDto) {
Post post = mapFromDtoToPost(postDto);
if (post.getId() == null) {
post.setDateCreated(Date.from(Instant.now()));
}
post.setLastUpdated(Date.from(Instant.now()));
postRepository.save(post);
}

@Transactional
public void deletePost(PostDto postDto) {
Post post = mapFromDtoToPost(postDto);
postRepository.delete(post);
}

@Transactional
public List<PostDto> showAllPosts() {
List<Post> posts = postRepository.findAll();
return posts.stream().map(this::mapFromPostToDto).collect(toList());
}
@Transactional
public PostDto readSinglePost(Long id) {
Post post = postRepository.findById(id).orElseThrow(() -> new
PostNotFoundException("For id " + id));
return mapFromPostToDto(post);
}


@Transactional
public void updatePost(PostDto existingPost) {
Post post = mapFromDtoToPost(existingPost);
postRepository.save(post);
```

```
}

private PostDto mapFromPostToDto(Post post) {
PostDto postDto = new PostDto();
postDto.setId(post.getId());
postDto.setImageUrl(post.getImageUrl());
postDto.setTitle(post.getTitle());
postDto.setContent(post.getContent());
postDto.setAuthor(post.getAuthor());
return postDto;
}

private Post mapFromDtoToPost(PostDto postDto) {
Post post = new Post();
post.setId(postDto.getId());
post.setImageUrl(postDto.getImageUrl());
post.setTitle(postDto.getTitle());
post.setContent(postDto.getContent());
post.setAuthor(postDto.getAuthor());//take this database later
post.setDateCreated(Date.from(Instant.now()));

return post;
}
}
```

1. `createPost`: This method creates a new post based on the provided `PostDto`. It maps the `PostDto` to a `Post` entity, sets the creation and last updated dates, and saves it using the `postRepository`.

2. `deletePost`: This method deletes a post based on the provided `PostDto`. It maps the `PostDto` to a `Post` entity and deletes it using the `postRepository`.

3. `showAllPosts`: This method retrieves all posts from the database. It uses the `postRepository` to fetch the posts, maps them to `PostDto` objects, and returns a list of `PostDto` representing all the posts.

4. `readSinglePost`: This method retrieves a single post based on the provided post ID. It uses the `postRepository` to find the post by its ID, maps it to a `PostDto`, and returns the corresponding `PostDto`.

5. `updatePost`: This method updates an existing post based on the provided `PostDto`. It maps the `PostDto` to a `Post` entity and saves it using the `postRepository`.

6. `mapFromPostToDto`: This private helper method maps a `Post` entity to a `PostDto` object, copying the relevant properties from the `Post` to the `PostDto`.

7. `mapFromDtoToPost`: This private helper method maps a `PostDto` object to a `Post` entity, copying the relevant properties from the `PostDto` to the `Post` and setting the creation date.

Overall, these methods provide functionality to create, delete, retrieve, and update posts in the system using the `PostRepository`. They use mapping methods to convert between `Post` entities and `PostDto` objects.

In Post Controller Class

```
@PutMapping("/posts/{id}")
public ResponseEntity<?> updatePost(@PathVariable Long id, @RequestBody PostDto
updatedPost) {
Optional<PostDto> optionalPost = Optional.ofNullable(postService.readSinglePost(id));
if (optionalPost.isPresent()) {
PostDto existingPost = optionalPost.get();

// Update the post with the new data
existingPost.setTitle(updatedPost.getTitle());
existingPost.setContent(updatedPost.getContent());

// Save the updated post
postService.updatePost(existingPost);

return new ResponseEntity<>(existingPost, HttpStatus.OK);
} else {
return new ResponseEntity<>("Post not found", HttpStatus.NOT_FOUND);
}
}

@DeleteMapping("/posts/{id}/delete")
public ResponseEntity deletePost(@PathVariable Long id) {
Optional<PostDto> optionalPost = Optional.ofNullable(postService.readSinglePost(id));
if (optionalPost.isPresent()) {
PostDto post = optionalPost.get();
postService.deletePost(post);
return new ResponseEntity<>(HttpStatus.OK);
} else {
return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
```

1. updatePost: This method is an HTTP PUT request mapping that updates an existing post with the provided id. It first checks if the post with the given ID exists by calling readSinglePost from the postService. If the post exists, it retrieves the existing post as existingPost. Then, it updates the title and content of the existingPost with the data from the updatedPost parameter. Finally, it calls updatePost from the postService to save the updated post and returns a response entity with the updated post and an HTTP status of OK (200). If the post is not found, it returns a response entity with an error message and an HTTP status of NOT_FOUND (404).
2. deletePost: This method is an HTTP DELETE request mapping that deletes a post with the provided id. It first checks if the post with the given ID exists by calling readSinglePost from the postService. If the post exists, it retrieves the post as post. Then, it calls deletePost from the postService to delete the post. Finally, it returns a response entity with an HTTP status of OK (200) indicating a successful deletion. If the post is not found, it returns a response entity with an HTTP status of NOT_FOUND (404).

These methods handle the update and deletion of posts by interacting with the postService and returning appropriate response entities based on the success or failure of the operations.

In the frontend part, the Add-post service .ts file was created similar to other services.

```
export class AddPostService {

 private baseUrl = "http://localhost:8080/api/posts";

 constructor(private httpClient: HttpClient) { }

 addPost(thePost: Post): Observable<any> {
  return this.httpClient.post(this.baseUrl, thePost);
 }

 getAllPosts(): Observable<Post[]> {
  return this.httpClient.get<GetResponse>(this.baseUrl).pipe(
    map((response: { _embedded: { posts: any; }; }) => response._embedded.posts)
  );
 }

 //for single post
 getPost(thePostId: number): Observable<Post> {

  //url based on id
```

```
  const postUrl = `${this.baseUrl}/${thePostId}`;

  return this.httpClient.get<Post>(postUrl);

 }

 updatePost(thePostId: number, updatedPost: Post): Observable<any> {
  const postUrl = `${this.baseUrl}/${thePostId}`;
  return this.httpClient.put(postUrl, updatedPost);
 }

 deletePost(thePostId: number): Observable<any> {
  const postUrl = `${this.baseUrl}/${thePostId}`;
  return this.httpClient.delete(postUrl);
 }

}
interface GetResponse {
 _embedded: {
  posts: Post[];
 }
}
```

The `AddPostService` class is responsible for handling the communication between the frontend application and the backend API for managing posts. Let's go through each method:

1. `addPost(thePost: Post)`: This method sends an HTTP POST request to the backend API at the specified `baseUrl` to add a new post. It sends the `thePost` object as the request body and returns an observable that emits the response from the server.

2. `getAllPosts()`: This method sends an HTTP GET request to the backend API at the `baseUrl` to retrieve all posts. It maps the response to extract the array of posts from the `_embedded` property and returns an observable that emits the array of posts.

3. `getPost(thePostId: number)`: This method sends an HTTP GET request to the backend API to retrieve a single post with the specified `thePostId`. It constructs the URL with the `thePostId` parameter and returns an observable that emits the post object.

4. `updatePost(thePostId: number, updatedPost: Post)`: This method sends an HTTP PUT request to the backend API to update the post with the specified `thePostId`. It constructs the URL with the `thePostId` parameter and sends the `updatedPost` object as the request body. It returns an observable that emits the response from the server.

5. `deletePost(thePostId: number)`: This method sends an HTTP DELETE request to the backend API to delete the post with the specified `thePostId`. It constructs the URL with the `thePostId` parameter and returns an observable that emits the response from the server.

The `GetResponse` interface defines the structure of the response received from the backend API when retrieving all posts. It specifies that the response should have an `_embedded` property containing an array of `Post` objects.

Overall, the `AddPostService` encapsulates the necessary methods to interact with the backend API for performing CRUD operations on posts in the frontend application.

Authentication

Login / Logout operations in the project were done using okta help.Authentication using Okta, the login and logout processes typically involve the following steps:

1. Login Process:
    - The user accesses the login page of the application.
    - The application redirects the user to the Okta authentication page.
    - The user enters their credentials (username and password) on the Okta authentication page.
    - Okta verifies the user's credentials and, if valid, generates an authentication token.
    - Okta redirects the user back to the application along with the authentication token.
    - The application receives the authentication token and validates it.
    - If the token is valid, the user is considered authenticated and can access protected resources in the application.
2. Logout Process:
    - The user initiates the logout process, usually by clicking on a "Logout" button or link.
    - The application sends a request to Okta to invalidate the user's session and revoke the authentication token.
    - Okta performs the necessary actions to terminate the user's session and invalidate the token.
    - Okta redirects the user back to the application, confirming that the logout process is complete.
    - The application clears any locally stored authentication-related information and updates the user interface accordingly.

5. Conclusion

In this project, we explored various aspects of developing an application with features such as product listing, cart management, user authentication, and post management. We examined different components, services, and APIs used to implement these functionalities.

For the product listing and filtering, we utilized the backend repository and frontend components to fetch and display products based on category and search keywords. The interactions between the frontend and backend were facilitated through HTTP requests.

The cart management was implemented using a CartService that provided methods for adding items to the cart, updating quantities, and removing items. The service maintained a list of cart items and computed the total price and quantity of the items.

User authentication was integrated using Okta, a popular authentication and authorization service. We discussed the login and logout processes, which involved user authentication through Okta and the validation of authentication tokens. Okta provided secure and reliable authentication services, enhancing the overall security of the application.

The post management functionality allowed users to create, update, and delete posts. The PostService was responsible for handling the CRUD operations and interacting with the backend repository to persist the data. We also examined how the frontend components interacted with the PostService to display and manipulate posts.

Overall, this project demonstrated the integration of various functionalities in a web application, including product listing, cart management, user authentication, and post management. By leveraging different technologies and services, we were able to create a robust and user-friendly application.

6.References

[1] https://www.w3schools.com/java/

[2] https://angular.io/

[3]https://spring.io/projects/spring-boot

[4] https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm

[5]https://www.mysql.com/

[6] https://www.okta.com/

[7] https://getbootstrap.com/

.