Project Documentation

Title: E-Commerce Web Application

1. Project Overview:

The E-Commerce Web Application is a platform that allows users to buy and sell products online. The application provides a user-friendly interface for customers to browse through a wide range of products, add them to their cart, and proceed with the purchase. Additionally, it offers an admin dashboard for managing products, categories, and user data.

2. Project Capabilities:

- User Registration and Login: The application allows users to create an account by providing their username, password, and email. After registration, users can log in to access personalized features.

- Product Listing: The platform displays a list of products with details such as product name, price, and an image. Customers can view product details before making a purchase.

- Shopping Cart: Customers can add products to their shopping cart while browsing. The cart keeps track of selected items and their quantities.

- Checkout and Payment: The application supports a secure checkout process, where users can review their cart contents, provide shipping information, and make payment using a secure payment gateway.

- Categories: The admin dashboard offers category management, enabling administrators to create and manage different product categories.

- User Management: Administrators can view user information, such as username, email, and role (Admin or Customer). They can also change user passwords.

- Purchase Reporting: The application generates purchase reports based on specified date ranges and selected categories.

3. Appearance:

The E-Commerce Web Application features a clean and intuitive user interface with a modern design. The frontend is designed using HTML, CSS, and JavaScript to provide a smooth and responsive user experience. The use of images for product representation enhances the visual appeal of the application.

4. User Interaction:

- Customer Interaction: Customers can interact with the application by browsing through the product list, viewing product details, adding products to the cart, and proceeding to checkout. They can also change their passwords and view their purchase history.

- Admin Interaction: Administrators interact with the application through the admin dashboard. They can manage products, categories, and user accounts. The admin dashboard provides a convenient way to monitor and control various aspects of the e-commerce platform.

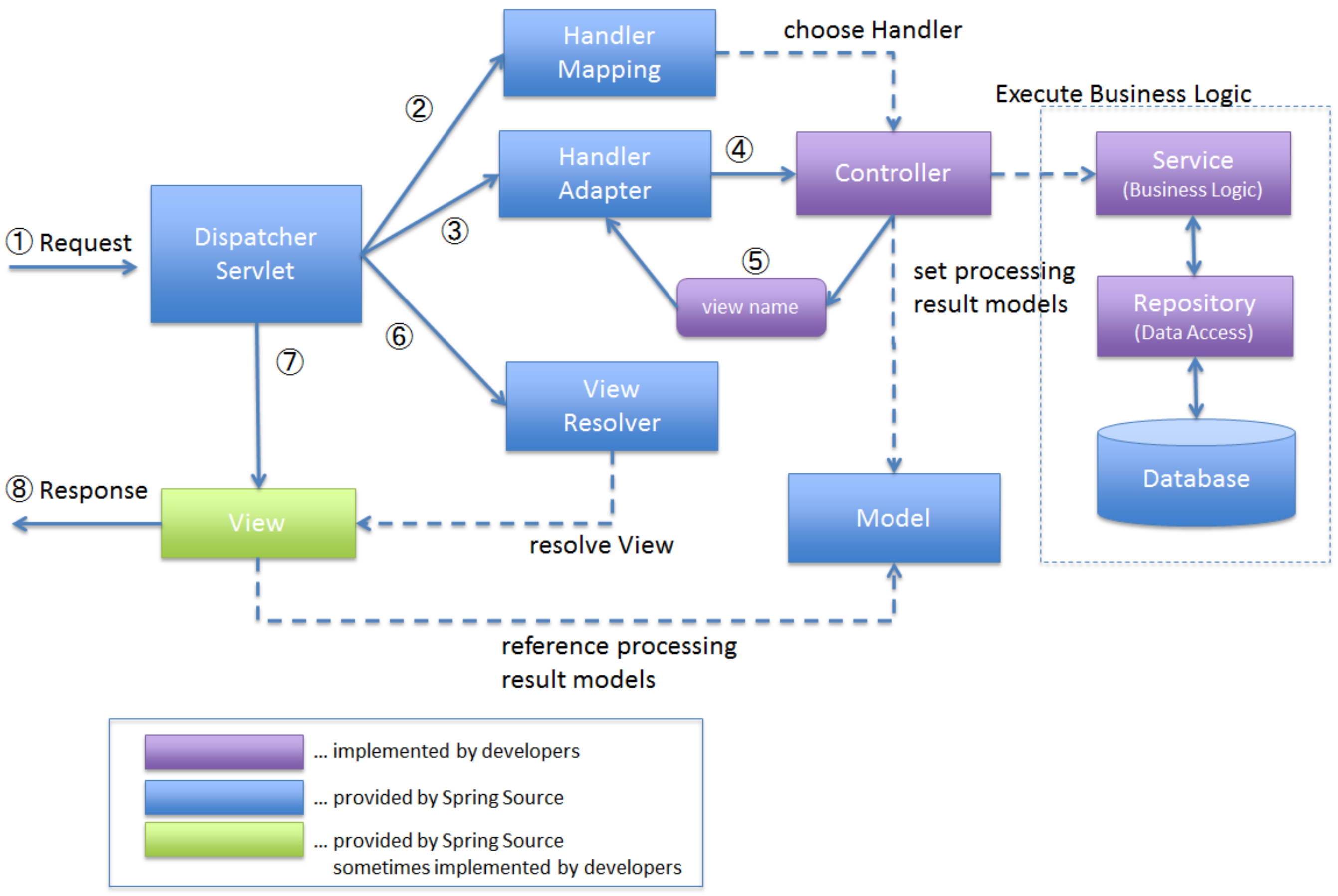
5. Technology Stack:

- Frontend: HTML, CSS, JavaScript, Bootstrap
- Backend: Java, Spring Framework (Spring Boot, Spring MVC, Spring Data JPA)
- Database: MySQL
- Templating: JSP (JavaServer Pages)
- Authentication: Spring Security
- Payment Gateway Integration: (External payment gateway service)
- Deployment: Apache Tomcat Server

6. Conclusion:

The E-Commerce Web Application is a feature-rich and user-friendly platform that provides customers with an easy way to purchase products online. With its robust admin dashboard, administrators can effectively manage the platform's products, categories, and users. The application's modern appearance and intuitive user interface contribute to a pleasant shopping experience for customers.

Note: This documentation provides an overview of the project's capabilities, appearance, and user interaction. Additional sections like project setup, database schema, and deployment instructions can be included in the complete project documentation.



Here is a breakdown of the XML elements and their meanings:

1. `project`: The root element of the POM.
2. `modelVersion`: Specifies the version of the POM model. In this case, it is set to 4.0.0.
3. `parent`: Indicates the parent project from which this project inherits configurations. The parent project is the Spring Boot starter parent with version 2.7.14.
4. `groupId`: Specifies the group or organization that owns the project. In this case, it is "com.simplilearn."
5. `artifactId`: The unique identifier for this project. It is named "SpringBootProjectSportyShoes."
6. `version`: The version of the project. The version is set to "0.0.1-SNAPSHOT," indicating that it is still under development.
7. `packaging`: Specifies the type of artifact to be built. Here, it is set to "war," indicating it will create a WAR file.
8. `name`: The name of the project.
9. `description`: A brief description of the project.
10. `properties`: Defines various properties used within the POM. The `java.version` property is set to "1.8," indicating that the project is using Java 8.
11. `dependencies`: Lists the dependencies required for the project. These are external libraries that the project relies on. The listed dependencies include Spring Boot web starter, JSTL, Spring Boot DevTools, Spring Boot Starter Tomcat, Spring Boot Data JPA, MySQL Connector/J, Tomcat Embed Jasper, H2 Database, and Spring Boot Starter Test.

12. `build`: Configures the build process for the project.
    - `plugins`: Specifies the plugins used during the build. In this case, there is a single plugin, the Spring Boot Maven Plugin, which is used to package the project into an executable JAR or WAR file.

2.

the main class of a Spring Boot web application named `SpringMvcApp2Application`. This class is the entry point of the application, and it uses Spring Boot annotations to configure and bootstrap the application.

This line specifies the package in which the `SpringMvcApp2Application` class is located. The class is in the package `org.simplilearn`.

These import statements bring in the required classes from the Spring Boot framework. `SpringApplication` is used to bootstrap the Spring application, and `@SpringBootApplication` is a composite annotation that combines `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`.

This annotation is applied to the class `SpringMvcApp2Application`, and it indicates that this class is the starting point of the Spring Boot application. It enables various features, such as component scanning and auto-configuration, which are essential for a Spring Boot application.

This is the class declaration. It defines the main class `SpringMvcApp2Application`.

This is the main method of the application. When the application is executed, this method will be called. Inside the `main` method, we use `SpringApplication.run(...)` to start the Spring Boot application. It takes two arguments: the class that contains the main method (`SpringMvcApp2Application.class`) and the command-line arguments (`args`).

When this application is executed, Spring Boot will initialize and start the embedded web server (such as Tomcat) and deploy the application. The application will be available at the default port (usually 8080) unless a custom port is configured.

3.

`ServletInitializer`, and it extends `SpringBootServletInitializer`. This class is typically used when you want to deploy your Spring Boot application as a WAR file and run it in an external servlet container like Apache Tomcat, instead of using the embedded servlet container that comes with Spring Boot.

This line specifies the package in which the `ServletInitializer` class is located. The class is in the package `org.simplilearn`.

These import statements bring in the required classes from the Spring Boot framework. `SpringBootServletInitializer` is a class provided by Spring Boot, and `SpringApplicationBuilder` is used for configuring the application builder.

This is the class declaration. It defines the `ServletInitializer` class, which extends `SpringBootServletInitializer`. By extending `SpringBootServletInitializer`, this class signals that the application should be deployed as a WAR file.

This is the overridden method `configure()`. When you extend `SpringBootServletInitializer`, you must override this method. This method is responsible for configuring the Spring application builder. In this case, the `application.sources(...)` method is called with `SpringMvcApp2Application.class` as an argument, indicating that the `SpringMvcApp2Application` class is the root configuration class of the Spring application.

When you build the WAR file and deploy it to an external servlet container like Apache Tomcat, the servlet container will use the `ServletInitializer` to initialize the Spring Boot application. The `configure()` method will then set up the application using the `SpringMvcApp2Application` class as the configuration source.

----

4. Entities

a. Cart

entity class named `Cart`, which represents a shopping cart for a user in a web application. The `Cart` entity is annotated with JPA annotations to map it to a corresponding database table. Let's break down the code:

package org.simplilearn.entities;

This line specifies the package in which the `Cart` class is located. The class is in the package `org.simplilearn.entities`.

import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import javax.persistence.Id; import javax.persistence.JoinColumn; import javax.persistence.ManyToOne; import javax.persistence.OneToOne;

These import statements bring in the required classes for JPA annotations and associations.

@Entity public class Cart {

The `@Entity` annotation marks the class as a JPA entity, indicating that instances of this class should be mapped to a database table.

@Id @GeneratedValue(strategy = GenerationType.IDENTITY) private int id;

The `@Id` annotation marks the primary key of the `Cart` entity. It corresponds to the primary key column in the database table. The `@GeneratedValue` annotation with the `GenerationType.IDENTITY` strategy indicates that the value for this ID will be automatically generated by the database (auto-increment) when a new `Cart` object is persisted.

private String item; private int qty; private int price;

These are attributes of the `Cart` entity, representing the item name, quantity, and price of items in the cart.

@OneToOne @JoinColumn(name = "user_id") private User user;

This `@OneToOne` association with `User` indicates that each cart is associated with one user. The `@JoinColumn` annotation specifies the foreign key column in the `Cart` table that references the primary key of the `User` table, linking the user to their cart.

@ManyToOne(fetch = FetchType.LAZY) @JoinColumn(name = "product_id") private Product product;

This `@ManyToOne` association with `Product` indicates that many carts can be associated with one product. The `@JoinColumn` annotation specifies the foreign key column in the `Cart` table that references the primary key of the `Product` table, linking a product to multiple carts.

public Cart() { // Default constructor (no-argument constructor) required by JPA. }

This is the default constructor, which is required by JPA for entity classes.

The class also includes getter and setter methods for all attributes, which are standard JavaBean conventions used for accessing and modifying the object's state.

`CartItem` class represents an item in a shopping cart with its details and a reference to the corresponding `Product` entity. This class is used to store information about the items in the cart before the actual purchase is made.

`Category` class represents a category entity with a unique identifier (`id`) and a name (`name`). This class is used to store and retrieve information about product categories in a product management system.

`Order` class represents an order entity with a unique identifier (`id`), an order date (`orderDate`), a user associated with the order (`user`), a list of order items (`orderItems`), and the total cart value (`totalCartValue`). This class is used to store and retrieve information about orders made by users in a web application.

`OrderItem` class represents an item within an order with a unique identifier (`id`), an item name (`itemName`), a quantity (`quantity`), a price (`price`), and an association with the corresponding `Order` entity. This class is used to store and retrieve information about items within an order in a web application.

`Product` class represents a product entity with a unique identifier (`pid`), a name (`name`), a price (`price`), an image URL (`imageUrl`), a creation date (`creationDate`), and an association with the corresponding `User` entity. This class is used to store and retrieve information about products in a web application's product management system.

`Purchase` class represents a purchase entity with a unique identifier (`id`), a user who made the purchase (`user`), a purchase date (`purchaseDate`), a category associated with the purchase (`category`), a list of purchase items (`purchaseItems`), and the total amount of the purchase (`totalAmount`). This class is used to store and retrieve information about purchases made by users in a web application.

`PurchaseItem` class represents an item within a purchase with a unique identifier (`id`), a purchase it belongs to (`purchase`), a product associated with the item (`product`), a quantity of the product in the purchase (`quantity`), and the price of the product in the purchase (`price`). This class is used to store and retrieve information about items within a purchase in a web application.

`User` class represents a user entity with a unique identifier (`id`), a username (`username`), a password (`password`), an email address (`email`), a role (`role`), a cart associated with the user (`cart`), and a list of products (`products`) and carts (`carts`) owned by the user. This class is used to store and retrieve information about users in a web application.

Controllers

This is a Spring MVC controller class named `UserController`. It handles various HTTP requests related to user management and purchase reporting. Let's go through the major functionalities of this controller:

1. `showHome()`: This method maps the root URL ("/") and returns the view name "home".
2. `showLogin()`: This method maps the "/showLogin" URL and returns the view name "login".
3. `showSignUp()`: This method maps the "/showSignUp" URL and returns the view name "signup".
4. `register()`: This method maps the "/signUp" URL and handles user registration. It takes parameters such as `username`, `password`, `email`, and `role`. It creates a `UserDto` object with the provided information and passes it to the `UserService` for user registration. The appropriate message is then added to the model based on the registration result, and the view name "signup" is returned.
5. `login()`: This method maps the "/login" URL and handles user login. It takes parameters `username`, `password`, and `HttpSession` object. It creates a `LoginDto` object with the provided username and password and uses the `UserService` to perform user login. If the login is successful, the user information is stored in the session, and the appropriate dashboard view ("adminDashboard" for admin role, "customerDashboard" for other roles) is returned. If the login fails, an error message is added to the model, and the view name "login" is returned.
6. `logout()`: This method maps the "/logout" URL and handles user logout. It invalidates the user's session and redirects to the root URL ("/").
7. `showChangePasswordForm()`: This method maps the "/changePassword" URL (GET) and returns the view name "changePassword" to show the change password form.
8. `changePassword()`: This method maps the "/changePassword" URL (POST) and handles the password change request. It takes parameters `oldPassword`, `newPassword`, `HttpSession`, and `Model`. It retrieves the user from the session and checks if the old password matches the user's current password. If it does, the user's password is updated to the new password using the `UserService`. Appropriate messages are added to the model based on the result, and the user is redirected to the "adminDashboard" page.
9. `showUsers()`: This method maps the "/showUsers" URL and retrieves a list of all users using the `UserService`. The list of users is added to the model, and the view name "viewUsers" is returned to display the user information.
10. `showPurchaseReportForm()`: This method maps the "/purchaseReport" URL (GET) and retrieves a list of all categories using the `CategoryService`. The list of categories is added to the model, and the view name "purchaseReport" is returned to display the purchase report form.
11. `getFilteredPurchaseReport()`: This method maps the "/purchaseReport" URL (POST) and handles the purchase report filtering based on start date, end date, and category. It takes parameters `startDate`, `endDate`, `categoryId`, and `Model`. It uses the `UserService` to retrieve the

filtered purchase report by date and category. The list of purchases is added to the model, and the view name "purchaseReport" is returned to display the filtered purchase report.

Please note that the behavior and functionality of this controller depend on the corresponding services (`UserService`, `ProductService`, and `CategoryService`) and entity classes (`User`, `Product`, `Category`, `Purchase`) used in the application. Additionally, the corresponding views (HTML templates) should be available for each view name returned by the controller methods.

`PurchaseController` class, which acts as a controller for handling the purchase-related operations in a web application. The controller handles HTTP requests and returns appropriate views for rendering.

Let's understand the main functionality of the `PurchaseController`:

1. Make Payment (`/makePayment`):
    - The `makePayment` method is invoked when the user initiates the payment process. It takes a `HttpSession` object as a parameter, which can be used to retrieve information stored in the session.
    - In the comments, it outlines the steps to process the payment:
        - Retrieve `cartItems` and `totalCartValue` from the session: Presumably, these are stored in the session during the user's shopping journey.
        - Perform payment processing: This step may involve integrating with a payment gateway or a payment service to process the payment transaction.
        - Create a `PurchaseDto` object: The `PurchaseDto` appears to be a data transfer object that holds data related to the purchase, such as the user and the purchased items.
        - Save the purchase: It calls the `purchaseService.savePurchase(purchaseDto)` to save the purchase details into the system. This would include information like the user who made the purchase, the purchased items, and the total amount paid.
    - After the successful purchase, the method returns the "payment_success" view. This view might be a page that confirms the successful completion of the purchase and thanks the user for their payment.
    - However, it's important to note that the actual implementation of the payment processing logic and the structure of the `PurchaseDto` class are not provided in the code snippet. This code mainly outlines the high-level steps involved in processing a purchase and suggests that payment processing and purchase details saving should be handled by the `PurchaseService` using the `purchaseService.savePurchase(purchaseDto)` method.
    - The exact implementation details may vary based on the specific requirements and the integration with external payment services, if any. The provided code snippet acts as a placeholder and indicates the logical flow of the purchase process within the `PurchaseController`. The missing details need to be implemented separately, and the actual payment processing and purchase storage would require integration with external services or databases.

----

code defines a `ProductController` class, which acts as a controller for handling product-related operations in a web application. The controller handles HTTP requests and returns appropriate views for rendering.

Let's understand the main functionalities of the `ProductController`:

1. Show Add Product Form (`/showAdd`):
    - The `showAddProduct` method returns the view for adding a new product. This view likely contains a form where administrators can enter details of a new product to be added.
2. Add Product (`/add`):
    - The `addProduct` method handles the process of adding a new product to the system. It takes input parameters for `name`, `price`, and `imageUrl`.
    - It converts the `price` parameter to an integer value and creates a `ProductDto` object with the provided input values.
    - It retrieves the logged-in user from the session using `HttpSession` and calls the `insertProduct` method of the `productService` to add the product to the system.
    - After successfully adding the product, it returns the "adminDashboard" view, which is the dashboard view for administrators.
3. Show Products (`/showProducts`):
    - The `showProducts` method retrieves all products from the `productService` and returns the "adminDashboard" view, which likely displays a list of products in the admin dashboard view.
4. Delete Product (`/delete/{pid}`):
    - The `deleteProduct` method handles the process of deleting a product. It takes the `pid` (product ID) as a path variable to identify the product to be deleted.
    - It calls the `deleteProduct` method of the `productService` to delete the product from the system.
    - After successfully deleting the product, it redirects to the "showProducts" URL, which will display the updated list of products in the admin dashboard view.

Note: The `ProductController` interacts with the `productService`, which handles business logic related to products, such as adding, retrieving, and deleting products. The code indicates that the view names used are "adminDashboard" and "addProduct," but the actual structure and content of these views are not provided in the code snippet.

Overall, the `ProductController` handles product-related operations, including displaying product lists, adding new products, and deleting existing products, while using the `ProductService` to manage product-related functionalities.

----

code defines a `CategoryController` class, which acts as a controller for handling category-related operations in a web application. The controller handles HTTP requests and returns appropriate views for rendering.

Let's understand the main functionalities of the `CategoryController`:

1. Show Add Category Form (`/showAddCategory`):
   - The `showAddCategoryForm` method returns the view for adding a new category. This view likely contains a form where administrators can enter the name of the new category to be created.
2. Add Category (`/addCategory`):
   - The `addCategory` method handles the process of adding a new category to the system. It takes the `name` parameter from the request parameters, which represents the name of the new category to be created.
   - It creates a `CategoryDto` object with the provided input name and calls the `createCategory` method of the `categoryService` to create the category in the system.
   - After creating the category, it sets a success or failure message in the model attribute (`msg`) based on the outcome of the category creation process.
   - The method then returns the "addCategory" view. If the category creation was successful, the success message will be displayed in this view; otherwise, the failure message will be displayed.
3. Show All Categories (`/showCategories`):
   - The `showCategories` method retrieves all categories from the `categoryService` and returns the "categories" view. This view likely displays a list of all categories available in the system.

Note: The `CategoryController` interacts with the `categoryService`, which handles business logic related to categories, such as creating and retrieving categories. The code indicates that the view names used are "addCategory" and "categories," but the actual structure and content of these views are not provided in the code snippet.

Overall, the `CategoryController` handles category-related operations, including displaying a form for creating a new category, creating a new category, and displaying a list of all categories, while using the `CategoryService` to manage category-related functionalities.

----

code defines a `CartController` class, which acts as a controller for handling cart-related operations in a web application. The controller handles HTTP requests and returns appropriate views for rendering.

Let's understand the main functionalities of the `CartController`:

1. Show Cart (`/showCart`):
   - The `showCart` method retrieves the user's cart items and calculates the total cart value using the `cartService`.
   - It adds the cart items and total cart value to the model attribute and returns the "cart" view to display the user's cart items.
2. Add to Cart (`/addToCart/{pid}`):
   - The `addToCart` method handles the process of adding a product to the user's cart.
   - It retrieves the product by its `pid` (product ID) from the `productService`.
   - It then creates a `CartModel` object with the product's name and price and adds the item to the user's cart using the `cartService`.
   - After adding the item to the cart, it redirects the user to the "showCart" page to view the updated cart.
3. Remove from Cart (`/removeFromCart/{pid}`):
   - The `removeFromCart` method handles the process of removing a product from the user's cart.
   - It removes the product with the given `pid` from the user's cart using the `cartService`.
   - After removing the item from the cart, it redirects the user to the "showCart" page to view the updated cart.
4. Checkout (`/checkout`):
   - The `checkout` method handles the process of checking out the user's cart items and creating an order.
   - It retrieves the user's cart items and calculates the total cart value using the `cartService`.
   - It creates an `Order` entity and sets the order date, user, total cart value, and order items (converted from cart items).
   - It then saves the order using the `orderService`.
   - After saving the order, it clears the user's cart and sets a session attribute (`checkoutSuccess`) to indicate successful checkout.
   - It redirects the user to the "makePayment" page to simulate the payment process.
5. Make Payment (`/makePayment`):
   - The `makePayment` method simulates the payment process for the order. In a real application, this is where you would integrate with a payment gateway to handle actual payments.
   - After simulating the payment, it redirects the user to the "checkout_success" page.
6. Checkout Success (`/checkout_success`):
   - The `checkoutSuccess` method handles the final step after successful checkout and payment.
   - It retrieves the user from the session and redirects the user to their respective dashboard based on their role (admin or regular user).
7. Purchase (`/purchase`):

- The `purchase` method displays the cart items and total cart value to the user before checkout. It's like a confirmation page for the user before proceeding to the actual checkout.

Overall, the `CartController` handles cart-related operations, including displaying the cart, adding/removing items from the cart, processing checkout and payment, and displaying a confirmation page before the final checkout. The controller uses the `ProductService`, `CartService`, and `OrderService` to manage cart and order-related functionalities.

Model (DTO)

The `CartModel` class is a simple POJO (Plain Old Java Object) that represents a cart item in the web application. It is used to store information about a product that is added to the user's cart. The class contains two fields: `item` and `price`.

Let's understand the fields and methods of the `CartModel` class:

1. Fields:
    - `item`: Represents the name or description of the product added to the cart.
    - `price`: Represents the price of the product added to the cart.
2. Constructors:
    - Default constructor: An empty constructor with no parameters. It is auto-generated.
    - Parameterized constructor: Takes the `item` and `price` as parameters to initialize the `CartModel` object with specific values.
3. Getters and Setters:
    - Getters: Methods to retrieve the values of the `item` and `price` fields.
    - Setters: Methods to set or update the values of the `item` and `price` fields.

The `CartModel` class is a lightweight data structure used to pass cart item information between different parts of the application. It allows the application to encapsulate product details conveniently and use them to perform cart-related operations. The class itself does not handle business logic; its main purpose is to represent a cart item's data.

The `CategoryDto` class is a Data Transfer Object (DTO) that represents the data needed to create or update a category in the web application. It is used as a container to transfer data between the presentation layer (controllers, views) and the service layer (business logic).

Let's understand the fields and methods of the `CategoryDto` class:

1. Fields:
   - `name`: Represents the name of the category.
2. Constructors:
   - Default constructor: An empty constructor with no parameters. It is auto-generated.
   - Parameterized constructor: Takes the `name` as a parameter to initialize the `CategoryDto` object with a specific name.
3. Getters and Setters:
   - Getters: Methods to retrieve the value of the `name` field.
   - Setters: Methods to set or update the value of the `name` field.

The `CategoryDto` class serves as a simple container for the category name data. It is typically used to transfer category-related information from the user interface (e.g., web form) to the service layer, where the actual category creation or update logic is performed. By using DTOs, we can decouple the presentation layer from the underlying data model and simplify the data exchange process.

The `LoginDto` class is a Data Transfer Object (DTO) used to transfer login-related data between the presentation layer (controllers, views) and the service layer (business logic) of the web application. It contains the necessary data to facilitate user login.

Let's understand the fields and methods of the `LoginDto` class:

1. Fields:
   - `username`: Represents the username provided by the user during login.
   - `password`: Represents the password provided by the user during login.
2. Constructors:
   - Default constructor: An empty constructor with no parameters. It is auto-generated.
   - Parameterized constructor: Takes both the `username` and `password` as parameters to initialize the `LoginDto` object with specific login credentials.
3. Getters and Setters:
   - Getters: Methods to retrieve the values of the `username` and `password` fields.
   - Setters: Methods to set or update the values of the `username` and `password` fields.

The `LoginDto` class serves as a container for the login credentials data, allowing the user interface to send the username and password securely to the backend service layer for authentication. By using DTOs, we can encapsulate the login information and pass it as an object, which helps keep the login process organized and maintainable. Additionally, DTOs facilitate data validation and handling before passing it to the service layer for further processing.

The `OrderDTO` class is a Data Transfer Object (DTO) used to transfer order-related data between the presentation layer (controllers, views) and the service layer (business logic) of the web application. It contains the necessary data to create an order, including user information and a list of order items.

Let's understand the fields and methods of the `OrderDTO` class:

1. Fields:
    - `userId`: Represents the unique identifier (ID) of the user who is placing the order.
    - `orderItems`: Represents a list of `OrderItemDTO` objects, each containing information about the products and quantities to be included in the order.
2. Getters and Setters:
    - Getters: Methods to retrieve the values of the `userId` and `orderItems` fields.
    - Setters: Methods to set or update the values of the `userId` and `orderItems` fields.

The `OrderDTO` class serves as a container for the data required to create an order. By encapsulating the order information in this DTO, the presentation layer can pass the order-related data to the service layer in a well-structured manner. This enables better data validation, handling, and processing in the service layer, leading to a more maintainable and organized application architecture.

The `OrderDTO` class is often used when handling data from the frontend (user interface) during the order creation process. It helps decouple the frontend and backend components, making it easier to manage and evolve the application over time.

The `OrderItemDTO` class is a Data Transfer Object (DTO) used to transfer information about individual order items between different layers of the application. It represents the data required to create an order item, such as the name of the item, the quantity, and the price.

Let's understand the fields and methods of the `OrderItemDTO` class:

1. Fields:
    - `itemName`: Represents the name of the item in the order.
    - `quantity`: Represents the quantity of the item in the order.
    - `price`: Represents the price of the item.
2. Getters and Setters:

- Getters: Methods to retrieve the values of the `itemName`, `quantity`, and `price` fields.
- Setters: Methods to set or update the values of the `itemName`, `quantity`, and `price` fields.

The `OrderItemDTO` class is used to encapsulate the data related to individual order items. When creating an order, the frontend (presentation layer) can use this DTO to collect information about each item in the order and then pass it to the backend (service layer) for further processing. The service layer can then use this DTO to create actual order items and perform any necessary calculations before saving the order to the database.

Using DTOs like `OrderItemDTO` helps in maintaining a clear separation between the data used in different layers of the application. It also provides a well-defined contract for data exchange, making it easier to handle and validate data during communication between different components. Additionally, DTOs can be useful in mapping data between different object types, such as converting entities to DTOs for presentation purposes.

The `ProductDto` class is a Data Transfer Object (DTO) used to transfer information about a product between different layers of the application. It represents the data required to create or update a product, such as the product's name, price, and image URL.

Let's understand the fields and methods of the `ProductDto` class:

1. Fields:
   - `name`: Represents the name of the product.
   - `price`: Represents the price of the product.
   - `imageUrl`: Represents the URL of the product's image.
2. Constructors:
   - `ProductDto()`: Default constructor.
   - `ProductDto(String name, int price, String imageUrl)`: Parameterized constructor to set the initial values of the fields.
3. Getters and Setters:
   - Getters: Methods to retrieve the values of the `name`, `price`, and `imageUrl` fields.
   - Setters: Methods to set or update the values of the `name`, `price`, and `imageUrl` fields.

The `ProductDto` class is used to encapsulate product-related data when transferring it between different layers of the application. For example, when a user submits a form to create or update a product, the data from the form can be collected by the frontend (presentation layer) and stored in a `ProductDto` object. This `ProductDto` can then be sent to the backend (service layer) for further processing, such as saving the product to the database.

Using DTOs like `ProductDto` helps in maintaining a clear separation between the data used in different layers of the application. It also provides a well-defined contract for data exchange, making it easier to handle and validate data during communication between different components. Additionally, DTOs can be useful in mapping data between different object types, such as converting DTOs to entities for database operations or vice versa.

The `PurchaseDto` class is another Data Transfer Object (DTO) used to transfer information related to a purchase between different layers of the application. It represents the data required to create a purchase, such as the user who made the purchase and the list of cart items that were purchased.

Let's understand the fields and methods of the `PurchaseDto` class:

1. Fields:
   - `user`: Represents the user who made the purchase. It is an instance of the `User` entity.
   - `cartItems`: Represents the list of cart items that were purchased. Each item is an instance of the `CartItem` entity.
2. Constructors:
   - `PurchaseDto()`: Default constructor.
   - `PurchaseDto(User user, List<CartItem> cartItems)`: Parameterized constructor to set the initial values of the fields.
3. Getters and Setters:
   - Getters: Methods to retrieve the values of the `user` and `cartItems` fields.
   - Setters: Methods to set or update the values of the `user` and `cartItems` fields.

The `PurchaseDto` class is used to encapsulate purchase-related data when transferring it between different layers of the application. For example, when a user completes a purchase, the data related to the purchase (such as the user making the purchase and the items in their cart) can be collected by the frontend (presentation layer) and stored in a `PurchaseDto` object. This `PurchaseDto` can then be sent to the backend (service layer) for further processing, such as creating a new `Purchase` entity in the database with the purchase details.

Using DTOs like `PurchaseDto` helps in maintaining a clear separation between the data used in different layers of the application. It also provides a well-defined contract for data exchange, making it easier to handle and validate data during communication between different components. Additionally, DTOs can be useful in mapping data between different object types, such as converting DTOs to entities for database operations or vice versa.

The `UserDto` class is a Data Transfer Object (DTO) used to transfer user-related information between different layers of the application. It represents the data required to create or retrieve user information, such as username, password, email, and role.

Let's understand the fields and methods of the `UserDto` class:

1. Fields:
   - `username`: Represents the username of the user.
   - `password`: Represents the password of the user.
   - `email`: Represents the email address of the user.
   - `role`: Represents the role of the user (e.g., "Customer" or "Admin").
2. Constructors:
   - `UserDto()`: Default constructor.
   - `UserDto(String username, String password, String email, String role)`: Parameterized constructor to set the initial values of the fields.
3. Getters and Setters:
   - Getters: Methods to retrieve the values of the `username`, `password`, `email`, and `role` fields.
   - Setters: Methods to set or update the values of the `username`, `password`, `email`, and `role` fields.

The `UserDto` class is used to encapsulate user-related data when transferring it between different layers of the application. For example, when a new user is signing up, the data entered by the user (such as username, password, email, and role) can be collected by the frontend (presentation layer) and stored in a `UserDto` object. This `UserDto` can then be sent to the backend (service layer) for further processing, such as creating a new `User` entity in the database with the user details.

Using DTOs like `UserDto` helps in maintaining a clear separation between the data used in different layers of the application. It also provides a well-defined contract for data exchange, making it easier to handle and validate data during communication between different components. Additionally, DTOs can be useful in mapping data between different object types, such as converting DTOs to entities for database operations or vice versa.

Repositories

The `CartRepository` interface extends the `JpaRepository` provided by Spring Data JPA, which enables the repository to perform basic CRUD (Create, Read, Update, Delete) operations on the `Cart` entity. It also defines two additional custom query methods to retrieve specific data from the `Cart` entity based on certain conditions.

Let's understand the methods defined in the `CartRepository`:

1. `findByUser(User user)`: This method retrieves a list of `Cart` entities associated with a specific `User`. It takes a `User` object as input and returns a list of `Cart` objects that belong to the given user.
2. `findByItemAndUser(String item, User user)`: This method retrieves a specific `Cart` entity associated with a given `User` and a specific item name. It takes two parameters, the item name (`String`) and the `User` object, and returns the corresponding `Cart` object if it exists.

These custom query methods are based on their method names and follow Spring Data JPA's query derivation approach. By naming the methods according to the conventions used in Spring Data JPA, you can avoid writing explicit JPQL (Java Persistence Query Language) queries.

For example, to find a user's cart items, you can call the `findByUser(User user)` method, and to find a specific item in a user's cart, you can call the `findByItemAndUser(String item, User user)` method.

Spring Data JPA will automatically generate the appropriate SQL queries based on the method names and the entity's relationships, saving you from writing boilerplate code for querying the database.

The `CategoryRepository` interface extends the `JpaRepository` provided by Spring Data JPA, which enables the repository to perform basic CRUD (Create, Read, Update, Delete) operations on the `Category` entity. Since it doesn't define any additional custom query methods, it will only have the default methods inherited from `JpaRepository`.

Here are the methods available in the `CategoryRepository`:

1. `save(S entity)`: Save a new or updated `Category` entity to the database.
2. `findById(ID id)`: Retrieve a `Category` entity by its primary key (`ID`).
3. `findAll()`: Retrieve all `Category` entities from the database.
4. `existsById(ID id)`: Check if a `Category` entity with a given primary key exists.
5. `count()`: Get the total count of `Category` entities in the database.
6. `deleteById(ID id)`: Delete a `Category` entity by its primary key.
7. `delete(T entity)`: Delete the specified `Category` entity from the database.
8. `deleteAll(Iterable<? extends T> entities)`: Delete all `Category` entities in the provided iterable.
9. `deleteAll()`: Delete all `Category` entities from the database.

By using Spring Data JPA's `JpaRepository`, you can leverage these methods to interact with the database without writing explicit SQL queries. If you need to add custom queries or more complex data retrieval operations specific to `Category` entities, you can define them in this interface by following Spring Data JPA's query derivation conventions.

The `OrderRepository` interface extends the `JpaRepository` provided by Spring Data JPA and defines a custom query method using the `@Query` annotation.

Here's what the `OrderRepository` interface provides:

1. Inherited Methods:
   - All the default CRUD methods such as `save()`, `findById()`, `findAll()`, `deleteById()`, etc., inherited from `JpaRepository<Order, Long>`.
2. Custom Query Method:
   - `List<Order> findByUser(User user)`: This method defines a custom query using the `@Query` annotation. It retrieves a list of `Order` entities based on the provided `User` entity. The query uses JPQL (Java Persistence Query Language), and the `?1` placeholder refers to the first parameter in the method, which is the `User` entity.

By using this custom query, you can retrieve a list of `Order` entities associated with a specific `User`, without the need to implement the query logic explicitly. Spring Data JPA will generate the appropriate SQL query based on the JPQL query defined in the `@Query` annotation.

The `ProductRepository` interface extends the `JpaRepository` provided by Spring Data JPA. It inherits all the basic CRUD (Create, Read, Update, Delete) methods from `JpaRepository`, eliminating the need to implement them manually.

Here's what the `ProductRepository` interface provides:

1. Inherited Methods:
   - All the default CRUD methods such as `save()`, `findById()`, `findAll()`, `deleteById()`, etc., inherited from `JpaRepository<Product, Integer>`.

By extending the `JpaRepository` with the `Product` entity and the type of its primary key (`Integer` in this case), you get access to a wide range of built-in methods that can be used to interact with the underlying database and perform various operations on the `Product` entity without the need to write custom queries or repository methods.

For example, you can use methods like `findAll()`, `findById()`, `save()`, and others to retrieve, update, and delete `Product` entities easily. Additionally, you can define custom queries using method names, and Spring Data JPA will automatically generate the necessary SQL queries based on the method names. This is possible through the Query Creation mechanism provided by Spring Data JPA.

The `PurchaseRepository` interface extends the `JpaRepository` provided by Spring Data JPA and adds some custom query methods to retrieve `Purchase` entities based on specific criteria.

Here's what the `PurchaseRepository` interface provides:

1. Inherited Methods:
   - All the default CRUD methods such as `save()`, `findById()`, `findAll()`, `deleteById()`, etc., inherited from `JpaRepository<Purchase, Long>`.
2. Custom Query Methods:
   - `findByPurchaseDateBetween(Date startDate, Date endDate)`: Retrieves a list of `Purchase` entities with purchase dates between the specified `startDate` and `endDate`.
   - `findByCategory(Category category)`: Retrieves a list of `Purchase` entities that belong to the specified `Category`.
   - `findByPurchaseDateBetweenAndCategory(Date startDate, Date endDate, Category category)`: Retrieves a list of `Purchase` entities with purchase dates between the specified `startDate` and `endDate` and also belong to the specified `Category`.

By declaring these custom query methods in the `PurchaseRepository`, Spring Data JPA will automatically generate the necessary SQL queries based on the method names. This allows you to easily retrieve specific data from the `Purchase` entity without having to write complex SQL queries manually. The repository methods abstract away the database interactions, making it easier to work with data in your application.

The `UserRepository` interface extends the `JpaRepository` provided by Spring Data JPA and adds some custom query methods to retrieve `User` entities based on specific criteria.

Here's what the `UserRepository` interface provides:

1. Inherited Methods:
   - All the default CRUD methods such as `save()`, `findById()`, `findAll()`, `deleteById()`, etc., inherited from `JpaRepository<User, Integer>`.
2. Custom Query Methods:
   - `findByUsernameAndPassword(String username, String password)`: Retrieves a `User` entity based on the specified `username` and `password`. This method is useful for implementing user login functionality, where you can find a user by their username and verify their password.
   - `findByUsername(String username)`: Retrieves a `User` entity based on the specified `username`. This method can be used to find a user by their username without requiring the password.

Services

The `CategoryService` interface declares the contract for managing categories in the application. It provides methods for creating categories, retrieving all categories, and finding a category by its ID.

Here's what each method in the `CategoryService` interface does:

1. `createCategory(CategoryDto categoryDto)`: This method is used to create a new category based on the provided `CategoryDto`. It takes a `CategoryDto` object as a parameter, which contains the necessary information to create the category (e.g., name). The method returns a `Category` object representing the newly created category.
2. `getAllCategories()`: This method retrieves a list of all categories available in the application. It returns a `List<Category>` containing all the category entities in the database.
3. `findById(Long categoryId)`: This method is used to find a category by its ID. It takes the category ID as a parameter and returns the corresponding `Category` entity. If no category with the given ID is found, the method may return `null` or throw an exception to indicate that the category does not exist.

The `CategoryService` interface provides a way to separate the business logic for managing categories from the data access layer (repositories). By defining this interface, you can implement different versions of the `CategoryService` (e.g., for testing or production) without changing the code that uses it. The actual implementation of the service will be provided in a class that implements this interface.

The `CartService` class provides various methods to manage the user's cart and cart items. Let's go through each method in detail:

1. `addToCart(User user, CartModel cartModel)`: This method is used to add an item to the user's cart. It takes a `User` object representing the user and a `CartModel` object containing information about the item to be added (e.g., item name and price). If the item already exists in the cart, the method updates the quantity and price of the existing cart item. If the item is not in the cart, it creates a new cart item and associates it with the user.
2. `getCartItems(User user)`: This method retrieves all the cart items associated with the given user. It returns a list of `CartItem` objects, which contain details of each item in the cart, including the product associated with it.
3. `removeFromCart(User user, int pid)`: This method removes a cart item from the user's cart based on the provided cart item ID (`pid`).
4. `calculateTotalCartValue(User user)`: This method calculates the total value of all cart items for the given user. It returns the sum of prices of all cart items.
5. `clearCart(User user)`: This method clears all cart items for the given user. It deletes all cart items associated with the user.

The `CartService` class acts as a bridge between the controller and the repository, providing business logic to manage the user's cart. It makes use of the `CartRepository` and

`UserRepository` to interact with the database and perform CRUD operations on the cart and cart items.

The `@Service` annotation marks this class as a Spring service, making it eligible for dependency injection and indicating that it provides business logic to other components of the application.

The `UserService` interface defines the contract for the service layer related to user management and purchase reports. Let's go through each method in detail:

1. `register(UserDto userDto)`: This method is used to register a new user. It takes a `UserDto` object containing the user's registration details (e.g., username, password, email, and role). It returns the registered `User` entity.
2. `updateUser(User user)`: This method is used to update the details of an existing user. It takes a `User` object representing the user to be updated and saves the changes to the database.
3. `login(LoginDto loginDto)`: This method is used for user login authentication. It takes a `LoginDto` object containing the user's login credentials (e.g., username and password). It returns the `User` entity if the login is successful, or null if the credentials are incorrect.
4. `getAllUsers()`: This method retrieves a list of all users in the system. It returns a list of `User` entities representing all registered users.
5. `getPurchaseReportByDateAndCategory(Date startDate, Date endDate, Category category)`: This method is used to generate a purchase report based on the specified date range and category. It takes the start date, end date, and a `Category` entity as parameters. It returns a list of `Purchase` entities that match the specified criteria.

The `UserService` interface abstracts the implementation of these methods, allowing for various implementations (e.g., using different databases or data access technologies) without changing the code in the controllers. By providing these methods, the service layer acts as an intermediary between the controller and the repository, encapsulating the business logic and allowing for separation of concerns in the application architecture.

The `UserServiceImpl` class is an implementation of the `UserService` interface. It provides the actual implementation for the methods defined in the interface. Let's review each method in the `UserServiceImpl` class:

1. `register(UserDto userDto)`: This method creates a new `User` entity based on the data provided in the `UserDto` object and saves it to the database using the `userRepository`. It returns the newly registered user.
2. `login(LoginDto loginDto)`: This method performs user authentication by checking the provided username and password against the stored user data in the database using the `userRepository`. If

the credentials match, the method returns the corresponding `User` entity; otherwise, it returns `null`.

3. `updateUser(User user)`: This method updates an existing `User` entity in the database using the `userRepository`.
4. `getAllUsers()`: This method retrieves a list of all users from the database using the `userRepository`.
5. `getPurchaseReportByDateAndCategory(Date startDate, Date endDate, Category category)`: This method retrieves a list of `Purchase` entities from the database that match the specified date range and category. It uses the `purchaseRepository` to execute the query.

Overall, the `UserServiceImpl` class encapsulates the business logic related to user registration, login, and retrieval of user data. Additionally, it provides a method to fetch purchase reports based on the specified date range and category, demonstrating the implementation of business logic related to purchase reports.

The `PurchaseService` interface defines a contract for handling purchase-related operations. It has a single method:

1. `savePurchase(PurchaseDto purchaseDto)`: This method is responsible for saving a purchase based on the data provided in the `PurchaseDto` object. The `PurchaseDto` class contains information about the user and the list of cart items for the purchase. The implementation of this method should convert the data from `PurchaseDto` to the corresponding entities (`Purchase`, `Order`, and `OrderItem`) and save them to the database.

The `PurchaseServiceImpl` class is the implementation of the `PurchaseService` interface. It handles the business logic for saving a purchase based on the data provided in the `PurchaseDto` object. Here's a breakdown of the implementation:

1. `purchaseRepository`: The service class is autowired with the `PurchaseRepository`, which allows it to interact with the database to save the purchase entity.
2. `savePurchase(PurchaseDto purchaseDto)`: This method is responsible for converting the data from the `PurchaseDto` object into the corresponding entities (`Purchase`, `PurchaseItem`) and saving them to the database.
3. `Purchase purchase = new Purchase()`: A new `Purchase` entity is created to represent the purchase transaction.
4. `purchase.setUser(purchaseDto.getUser())`: The user associated with the purchase is set based on the `User` object provided in the `PurchaseDto`.
5. `purchase.setPurchaseDate(new Date())`: The current date is set as the purchase date.

6. `List<PurchaseItem> purchaseItems = purchaseDto.getCartItems().stream().map(cartItem -> {...}).collect(Collectors.toList())`: The cart items from the `PurchaseDto` are converted into a list of `PurchaseItem` entities. For each `CartItem` in the list, a new `PurchaseItem` is created and associated with the purchase.
7. `purchase.setPurchaseItems(purchaseItems)`: The list of `PurchaseItem` entities is set as the purchase items for the purchase.
8. `double totalAmount = purchaseItems.stream().mapToDouble(item -> item.getQuantity() * item.getPrice()).sum()`: The total purchase amount is calculated by multiplying the quantity and price of each `PurchaseItem` and summing them up.
9. `purchase.setTotalAmount(totalAmount)`: The total purchase amount is set in the `Purchase` entity.
10. `return purchaseRepository.save(purchase)`: The `Purchase` entity is saved to the database using the `purchaseRepository.save` method, and the saved entity is returned.

The `ProductService` interface defines the contract for the product-related business logic and data operations. Here's an explanation of the methods declared in the `ProductService` interface:

1. `void insertProduct(ProductDto productDto, User user)`: This method is used to insert a new product into the system. It takes a `ProductDto` object containing the product details and a `User` object representing the user who is adding the product. The implementation of this method should handle the creation of a new `Product` entity using the data from the `ProductDto` object and associate it with the provided `User`. The new product is then saved to the database.
2. `void deleteProduct(int pid)`: This method is used to delete a product from the system based on its product ID (`pid`). The implementation of this method should locate the product with the given ID, and if found, delete it from the database.
3. `List<Product> getAll()`: This method retrieves a list of all products available in the system. The implementation of this method should fetch all the products from the database and return them as a list of `Product` entities.
4. `Product getProduct(int pid)`: This method retrieves a specific product based on its product ID (`pid`). The implementation of this method should find the product with the provided ID in the database and return it as a `Product` entity.

The `ProductServiceImpl` class implements the `ProductService` interface, providing the actual implementation for the product-related business logic and data operations. Here's an explanation of the methods in the `ProductServiceImpl` class:

1. `void insertProduct(ProductDto productDto, User user)`: This method inserts a new product into the system. It takes a `ProductDto` object containing the product details and a `User` object representing the user who is adding the product. The implementation of this method fetches the

user from the database using the provided username and password (assuming that the `UserRepository` has a method to find a user by username and password). Then, it creates a new `Product` entity using the data from the `ProductDto` object and associates it with the retrieved user. Finally, the new product is saved to the database.

2. `void deleteProduct(int pid)`: This method deletes a product from the system based on its product ID (`pid`). The implementation of this method deletes the product with the provided ID from the database using the `ProductRepository`'s `deleteById()` method.
3. `List<Product> getAll()`: This method retrieves a list of all products available in the system. The implementation of this method fetches all the products from the database using the `ProductRepository`'s `findAll()` method and returns them as a list of `Product` entities.
4. `Product getProduct(int pid)`: This method retrieves a specific product based on its product ID (`pid`). The implementation of this method fetches the product with the provided ID from the database using the `ProductRepository`'s `findById()` method and returns it as a `Product` entity. If no product with the given ID is found, it returns `null`.

The `OrderService` interface defines the contract for handling order-related business logic and data operations. Here's an explanation of the methods in the `OrderService` interface:

1. `void saveOrder(Order order)`: This method is responsible for saving an order to the system. It takes an `Order` object representing the order to be saved. The implementation of this method should persist the order information to the database using the appropriate repository.
2. `List<Order> getOrdersByUser(User user)`: This method retrieves a list of orders associated with a specific user. It takes a `User` object representing the user whose orders are to be retrieved. The implementation of this method should fetch all the orders for the provided user from the database and return them as a list of `Order` entities.

By defining these methods in the `OrderService` interface, the application can provide different implementations for these methods depending on the underlying data storage technology or business requirements. For example, the actual implementation of the `OrderService` could use a `JpaRepository` to interact with a relational database or any other data access mechanism suitable for the application.

The `OrderServiceImpl` class is the concrete implementation of the `OrderService` interface. It provides the actual implementation for the methods defined in the interface to handle order-related business logic and data operations. Here's an explanation of the methods in the `OrderServiceImpl` class:

1. `void saveOrder(Order order)`: This method is responsible for saving an order to the system. It takes an `Order` object representing the order to be saved. The implementation uses the `orderRepository` to persist the order information to the database.
2. `List<Order> getOrdersByUser(User user)`: This method retrieves a list of orders associated with a specific user. It takes a `User` object representing the user whose orders are to be retrieved. The implementation uses the `orderRepository` to fetch all the orders for the provided user from the database and returns them as a list of `Order` entities.

By using the `OrderRepository` as a data access mechanism, the `OrderServiceImpl` can interact with the underlying database to perform CRUD (Create, Read, Update, Delete) operations on the `Order` entity. The `@Autowired` annotation in the constructor indicates that the `OrderRepository` dependency will be automatically injected by Spring when creating an instance of the `OrderServiceImpl` class.

The `CategoryServiceImpl` class is the concrete implementation of the `CategoryService` interface. It provides the actual implementation for the methods defined in the interface to handle category-related business logic and data operations. Let's go through the methods in the `CategoryServiceImpl` class:

1. `Category createCategory(CategoryDto categoryDto)`: This method is responsible for creating a new category in the system. It takes a `CategoryDto` object representing the category to be created. The implementation creates a new `Category` entity, sets its name based on the `CategoryDto`, and then uses the `categoryRepository` to save the category to the database.
2. `List<Category> getAllCategories()`: This method retrieves a list of all categories available in the system. The implementation uses the `categoryRepository` to fetch all categories from the database and returns them as a list of `Category` entities.
3. `Category findById(Long categoryId)`: This method retrieves a specific category by its ID. It takes a `Long` value representing the ID of the category to be retrieved. The implementation uses the `categoryRepository` to find the category by its ID in the database. If the category is found, it returns the `Category` entity; otherwise, it returns `null`.

By using the `CategoryRepository` as a data access mechanism, the `CategoryServiceImpl` can interact with the underlying database to perform CRUD (Create, Read, Update, Delete) operations on the `Category` entity. The `@Autowired` annotation in the constructor indicates that the `CategoryRepository` dependency will be automatically injected by Spring when creating an instance of the `CategoryServiceImpl` class.

This implementation allows the application to manage and retrieve categories from the database using the `CategoryService` interface without worrying about the underlying database operations. It promotes loose coupling between the business logic and data access layers, making the application more modular and maintainable

---

JSP

The provided code appears to be an HTML page that will be displayed when the checkout process is successful. This page serves as a confirmation to the user that their payment was successful, and their order has been placed. Let's go through the contents of the HTML page:

1. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
2. `<html>`: This tag marks the beginning of the HTML document.
3. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
4. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1, which is a character encoding for the Latin-1 character set.
5. `<title>Checkout Success</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab.
6. `</head>`: Marks the end of the `<head>` section.
7. `<body>`: The `<body>` element contains the visible content of the HTML page.
8. `<h1>Checkout Successful</h1>`: This is a level 1 heading, displaying the text "Checkout Successful" in a larger font.
9. `<p>Thank you for your order!</p>`: This is a paragraph tag, displaying the text "Thank you for your order!".
10. `<p>Your payment was successful, and your order has been placed.</p>`: Another paragraph tag displaying a confirmation message.
11. `<p>Your order will be processed and delivered soon.</p>`: Another paragraph tag with information about the order processing.
12. `<a href="/home">Back to Home</a>`: This is an anchor tag (`<a>`) with an `href` attribute set to "/home". This means that when the user clicks on this link, they will be redirected to the "/home" URL. Presumably, this is the link to return to the home page after the successful checkout.
13. `</body>`: Marks the end of the `<body>` section.
14. `</html>`: Marks the end of the HTML document.

Overall, this HTML page provides a clear and concise message to the user that their checkout was successful and includes a link to return to the home page. It is a simple and effective way to provide feedback to the user after a successful purchase.

The provided code appears to be an HTML page that allows users to change their password. Let's go through the contents of the HTML page:

1. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.

2. `<html>`: This tag marks the beginning of the HTML document.
3. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
4. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1, which is a character encoding for the Latin-1 character set.
5. `<title>Change Password</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab as "Change Password".
6. `</head>`: Marks the end of the `<head>` section.
7. `<body>`: The `<body>` element contains the visible content of the HTML page.
8. `<h1>Change Password</h1>`: This is a level 1 heading, displaying the text "Change Password" in a larger font, indicating the purpose of the page.
9. `<form method="post" action="./changePassword">`: This is an HTML form that allows users to enter their old password and new password. The form is set to use the "post" method and sends the data to the URL "./changePassword" when submitted.
10. `<label for="oldPassword">Old Password:</label>`: This line creates a label for the input field where users can enter their old password. The "for" attribute is associated with the "id" attribute of the input field.
11. `<input type="password" name="oldPassword" required>`: This is an input field of type "password" where users can enter their old password. The "name" attribute is set to "oldPassword", and the "required" attribute specifies that the field must be filled out before the form can be submitted.
12. `<br>`: This line creates a line break, moving the next content to a new line.
13. `<label for="newPassword">New Password:</label>`: This line creates a label for the input field where users can enter their new password. The "for" attribute is associated with the "id" attribute of the input field.
14. `<input type="password" name="newPassword" required>`: This is an input field of type "password" where users can enter their new password. The "name" attribute is set to "newPassword", and the "required" attribute specifies that the field must be filled out before the form can be submitted.
15. `<br>`: This line creates a line break, moving the next content to a new line.
16. `<input type="submit" value="Change Password">`: This is a submit button that users can click to submit the form and change their password. The text on the button is "Change Password".
17. `</form>`: Marks the end of the HTML form.
18. `</body>`: Marks the end of the `<body>` section.
19. `</html>`: Marks the end of the HTML document.

Overall, this HTML page provides a simple form for users to change their password. The form includes input fields for the old password and the new password, along with a submit button to trigger the password change action. This page is user-friendly and allows users to easily update their password.

The provided code appears to be a JSP (JavaServer Pages) file that generates an HTML page displaying a table of categories. Let's go through the contents of the JSP file:

1. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
2. `<html>`: This tag marks the beginning of the HTML document.
3. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
4. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1, which is a character encoding for the Latin-1 character set.
5. `<title>Categories</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab as "Categories".
6. `</head>`: Marks the end of the `<head>` section.
7. `<body>`: The `<body>` element contains the visible content of the HTML page.
8. `<h1>Categories</h1>`: This is a level 1 heading, displaying the text "Categories" in a larger font, indicating the purpose of the page.
9. `<table border="1">`: This line starts the definition of an HTML table with a border attribute set to "1", which adds a border around the table cells.
10. `<tr>`: This represents a table row, and the following content will be placed inside a table cell.
11. `<th>ID</th>` and `<th>Name</th>`: These are table header cells (`<th>`) that display the column headers "ID" and "Name" respectively.
12. `<%@ page import="java.util.List" %>` and `<%@ page import="org.simplilearn.entities.Category" %>`: These are JSP import directives that allow importing Java classes to use within the JSP page.
13. `<% List<Category> categories = (List<Category>) request.getAttribute("categories"); %>`: This JSP scriptlet retrieves the "categories" attribute from the request object. It assumes that the "categories" attribute is a list of `Category` objects, and it assigns the list to a local variable `categories`.
14. `<% for (Category category : categories) { %>` and `<% } %>`: These are JSP scriptlets that iterate through the list of categories using a for-each loop. For each `Category` object in the list, the content inside the loop will be executed.
15. `<td><%= category.getId() %></td>` and `<td><%= category.getName() %></td>`: These are JSP expressions that output the ID and name of each category as table data (`<td>`).
16. `</tr>`: Marks the end of the table row.
17. `</table>`: Marks the end of the HTML table.
18. `</body>`: Marks the end of the `<body>` section.
19. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page with a table displaying the categories. It expects a list of `Category` objects to be provided as an attribute named "categories" in the request, and it uses JSP scriptlets and expressions to iterate through the list and display the ID and name of each category in the table.

This JSP file is used to generate an HTML page that displays the contents of the user's cart. It utilizes JSTL (JavaServer Pages Standard Tag Library) to iterate through the cart items and display them in a table. Let's go through the contents of the JSP file:

1. `<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>`: This line sets the language and character encoding for the JSP page.
2. `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>`: This taglib directive imports the JSTL core library, allowing the usage of JSTL tags in the JSP page.
3. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
4. `<html>`: This tag marks the beginning of the HTML document.
5. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
6. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1, which is a character encoding for the Latin-1 character set.
7. `<title>Cart</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab as "Cart".
8. `</head>`: Marks the end of the `<head>` section.
9. `<body>`: The `<body>` element contains the visible content of the HTML page.
10. `<h1>Your Cart</h1>`: This is a level 1 heading, displaying the text "Your Cart" in a larger font, indicating the purpose of the page.
11. `<table border="1">`: This line starts the definition of an HTML table with a border attribute set to "1", which adds a border around the table cells.
12. `<tr>`: This represents a table row, and the following content will be placed inside a table cell.
13. `<th>Item Name</th>`, `<th>Quantity</th>`, `<th>Price</th>`, and `<th>Remove</th>`: These are table header cells (`<th>`) that display the column headers "Item Name", "Quantity", "Price", and "Remove" respectively.
14. `<c:forEach var="item" items="${cartItems}">`: This JSTL forEach tag iterates over the "cartItems" list and assigns each item to the variable "item" within the loop. `${cartItems}` is an expression that refers to the list of cart items provided as an attribute in the request.
15. `<td>${item.name}</td>`, `<td>${item.quantity}</td>`, and `<td>${item.price}</td>`: These are JSP expressions that display the name, quantity, and price of each cart item as table data (`<td>`).
16. `<td><a href="./removeFromCart/${item.pid}">Remove</a></td>`: This line generates a table data cell with a link to remove the item from the cart. `${item.pid}` is an expression that refers to the product ID of each cart item, and it will be used to construct the URL for removing the item.
17. `</tr>`: Marks the end of the table row.
18. `</table>`: Marks the end of the HTML table.
19. `<p>Total Cart Value: ${totalCartValue}</p>`: This line displays the total cart value using the JSP expression `${totalCartValue}`. `${totalCartValue}` is expected to be provided as an attribute in the request.
20. `<a href="./checkout">Proceed to Checkout</a>`: This line generates a link to the checkout page, allowing the user to proceed with the checkout process.
21. `</body>`: Marks the end of the `<body>` section.
22. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that displays the contents of the user's cart in a table format. The table includes columns for item name, quantity, price, and a link to remove each item from the cart. The total cart value is also displayed, and a link is provided to proceed to the checkout page.

This JSP file is used to generate an HTML page for the Admin Dashboard. The dashboard displays various options and information for an admin user. Let's go through the contents of the JSP file:

1. `<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>`: This line sets the language and character encoding for the JSP page.
2. `<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>`: This taglib directive imports the JSTL core library, allowing the usage of JSTL tags in the JSP page.
3. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
4. `<html>`: This tag marks the beginning of the HTML document.
5. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
6. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1, which is a character encoding for the Latin-1 character set.
7. `<title>Insert title here</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab as "Insert title here".
8. `</head>`: Marks the end of the `<head>` section.
9. `<body>`: The `<body>` element contains the visible content of the HTML page.
10. `<% if (session.getAttribute("user")!=null) { %>`: This scriptlet checks if there is an authenticated user in the session. If an authenticated user is present, it displays the content inside the if block.
11. `<h1>Hi ${sessionScope.user.username}, Welcome to Admin Dashboard</h1>`: This line displays a greeting message with the username of the authenticated admin user.
12. `<a href="./showAdd">Add Product</a>`, `<a href="./showProducts">Show Products</a>`, `<a href="./showCategories">Manage Categories</a>`, `<a href="showAddCategory">Add Category</a>`, `<a href="./showUsers">View Users</a>`, `<a href="./purchaseReport">Purchase Report</a>`, `<a href="./changePassword">Change Password</a>`, `<a href="./logout">Logout</a>`: These are links/buttons that provide various options for the admin user to navigate and perform different actions.
13. `<table border="1">`: This line starts the definition of an HTML table with a border attribute set to "1", which adds a border around the table cells.
14. `<tr>`, `<th>Name</th>`, `<th>Price</th>`, `<th>Image</th>`, and `<tr>`: These tags represent a table row and table header cells (`<th>`) that display the column headers "Name", "Price", and "Image" respectively.
15. `<c:forEach var="product" items="${products }">`: This JSTL forEach tag iterates over the "products" list and assigns each product to the variable "product" within the loop. `${products}` is an expression that refers to the list of products provided as an attribute in the request.

16. `<td>${product.name }</td>`, `<td>${product.price }</td>`, and `<td><img style="width: 100px; height: 100px" src="${product.imageUrl}" alt="${product.name}"/></td>`: These are JSP expressions that display the name, price, and image of each product as table data (`<td>`). The image is displayed with a fixed width and height.
17. `<td><a href="./delete/${product.pid }">Delete</a></td>`: This line generates a table data cell with a link to delete the corresponding product. `${product.pid}` is an expression that refers to the product ID of each product, and it will be used to construct the URL for deleting the product.
18. `</c:forEach>`: Marks the end of the forEach loop.
19. `</table>`: Marks the end of the HTML table.
20. `<% } else { %>`, `<%`, and `<% } %>`: These scriptlet blocks handle the case when there is no authenticated user. It redirects the user to the login page with a message to login first.
21. `</body>`: Marks the end of the `<body>` section.
22. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page for the Admin Dashboard. It displays the username of the authenticated admin user along with various options to add products, manage categories, view users, generate a purchase report, change the password, and logout. The dashboard also shows a table listing the products with their names, prices, and images, along with a link to delete each product. If no user is authenticated, the page redirects to the login page with a message to log in first.

This JSP file represents a form that allows the user to add a new product. Let's go through the contents of the JSP file:

1. `<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>`: This line sets the language and character encoding for the JSP page.
2. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
3. `<html>`: This tag marks the beginning of the HTML document.
4. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
5. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1, which is a character encoding for the Latin-1 character set.
6. `<title>Insert title here</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab as "Insert title here".
7. `</head>`: Marks the end of the `<head>` section.
8. `<body>`: The `<body>` element contains the visible content of the HTML page.
9. `<form action="./add" method="post">`: This line starts an HTML form with the "action" attribute set to "./add" and the "method" attribute set to "post". When the form is submitted, it will be sent to the URL "/add" using the HTTP POST method.

10. `Name: <input type="text" name="name"><br>`: This line creates a text input field for the user to enter the name of the product. The "name" attribute of the input field is set to "name".
11. `Price: <input type="text" name="price"><br>`: This line creates a text input field for the user to enter the price of the product. The "name" attribute of the input field is set to "price".
12. `ImageUrl: <input type="text" name="imageUrl"><br>`: This line creates a text input field for the user to enter the URL of the product's image. The "name" attribute of the input field is set to "imageUrl".
13. `<input type="submit" value="Submit">`: This line creates a submit button that the user can click to submit the form.
14. `</form>`: Marks the end of the HTML form.
15. `</body>`: Marks the end of the `<body>` section.
16. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page with a form that allows the user to add a new product. The form contains input fields for the name, price, and image URL of the product. When the user fills in the details and clicks the "Submit" button, the form will be submitted to the "/add" URL using the HTTP POST method, and the product details will be processed on the server side.

This is a simple JSP file that represents a form to add a new category. Let's go through the contents of the JSP file:

1. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
2. `<html>`: This tag marks the beginning of the HTML document.
3. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
4. `<meta charset="UTF-8">`: This line sets the character encoding of the page to UTF-8, which is a widely used character encoding that supports a wide range of characters from different languages.
5. `<title>Add Category</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab as "Add Category".
6. `</head>`: Marks the end of the `<head>` section.
7. `<body>`: The `<body>` element contains the visible content of the HTML page.
8. `<h1>Add Category</h1>`: This line displays the heading "Add Category" on the page.
9. `<form action="addCategory" method="post">`: This line starts an HTML form with the "action" attribute set to "addCategory" and the "method" attribute set to "post". When the form is submitted, it will be sent to the URL "/addCategory" using the HTTP POST method.
10. `Name: <input type="text" name="name" required>`: This line creates a text input field for the user to enter the name of the new category. The "name" attribute of the input field is set to "name", and the "required" attribute makes it a required field, meaning the user must enter a value before submitting the form.

11. `<input type="submit" value="Add Category">`: This line creates a submit button that the user can click to add the new category. When the button is clicked, the form will be submitted to the "addCategory" URL using the HTTP POST method.
12. `</form>`: Marks the end of the HTML form.
13. `<p>${msg}</p>`: This line displays the value of the "msg" variable, which may be set in the server-side code and passed to the JSP file. It can be used to display any messages or feedback to the user.
14. `</body>`: Marks the end of the `<body>` section.
15. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page with a form that allows the user to add a new category. The form contains an input field for the name of the category, and a "Add Category" button to submit the form. If there is any message or feedback from the server-side code (stored in the "msg" variable), it will be displayed as a paragraph on the page.

This is a JSP file that represents the checkout page for a shopping cart. Let's go through the contents of the JSP file:

1. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
2. `<html>`: This tag marks the beginning of the HTML document.
3. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
4. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1.
5. `<title>Checkout</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab as "Checkout".
6. `</head>`: Marks the end of the `<head>` section.
7. `<body>`: The `<body>` element contains the visible content of the HTML page.
8. `<h1>Checkout</h1>`: This line displays the heading "Checkout" on the page.
9. `<table border="1">`: This line starts an HTML table with a border attribute set to "1".
10. `<tr>`, `<th>`: These HTML tags define table rows and table headers respectively. The table headers are "Item Name", "Quantity", and "Price".
11. `<%@ page import="java.util.List" %>`: This directive imports the `java.util.List` class, allowing us to use the `List` data structure in the JSP.
12. `<%@ page import="org.simplilearn.entities.CartItem" %>`: This directive imports the `org.simplilearn.entities.CartItem` class, allowing us to use the `CartItem` class in the JSP.
13. `<% ... %>`: This is a scriptlet, which allows us to write Java code within the JSP page. In this case, we are iterating over the list of `CartItem` objects stored in the request attribute "cartItems".
14. `<%= item.getName() %>`, `<%= item.getQuantity() %>`, `<%= item.getPrice() %>`: These expressions are used to output the values of the `CartItem` properties (name, quantity, and price) in the table rows.
15. `</table>`: Marks the end of the HTML table.

16. `<p>Total Cart Value: <%= request.getAttribute("totalCartValue") %></p>`: This line displays the total cart value by retrieving it from the request attribute "totalCartValue" using the `<%= ... %>` expression.
17. `<a href="./makePayment">Proceed to Payment</a>`: This line creates a hyperlink with the text "Proceed to Payment". The URL of the hyperlink points to "./makePayment", which represents the URL where the user can make the payment.
18. `</body>`: Marks the end of the `<body>` section.
19. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that displays the items in the shopping cart along with their quantities and prices. It also shows the total cart value. The user can proceed to the payment page by clicking the "Proceed to Payment" link, which redirects to the URL "/makePayment".

This is a JSP file that represents the customer dashboard page for a shopping website. Let's go through the contents of the JSP file:

1. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
2. `<html>`: This tag marks the beginning of the HTML document.
3. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
4. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1.
5. `<title>Insert title here</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab.
6. `</head>`: Marks the end of the `<head>` section.
7. `<body>`: The `<body>` element contains the visible content of the HTML page.
8. `<h1>Hi ${sessionScope.user.username}, Welcome to Customer Dashboard</h1>`: This line displays a personalized greeting message to the user, showing their username from the session scope.
9. `<table border="1">`: This line starts an HTML table with a border attribute set to "1".
10. `<tr>`, `<th>`: These HTML tags define table rows and table headers respectively. The table headers are "Name", "Price", and "Image".
11. `<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>`: This taglib directive imports the JSTL core library, which allows us to use JSTL tags and expressions in the JSP.
12. `<c:forEach var="product" items="${products }">`: This JSTL forEach loop iterates over the list of products stored in the request attribute "products".
13. `<td>${product.name }</td>`, `<td>${product.price }</td>`: These expressions output the name and price of each product in the table cells.
14. `<td><img style="width: 100px; height: 100px" src="${product.imageUrl}" alt="${product.name}" /></td>`: This line displays the product image as an HTML image tag with a width and height of 100px. The image source is retrieved from the "imageUrl" property of each product, and the "alt" attribute is set to the product name.

15. `<td><a href="./addToCart/${product.pid}">Add to Cart</a></td>`: This line creates a hyperlink with the text "Add to Cart". The URL of the hyperlink points to "./addToCart/${product.pid}", where "${product.pid}" is the product ID. This link allows the user to add the product to their shopping cart.
16. `</tr>`: Marks the end of the table row.
17. `</c:forEach>`: Marks the end of the forEach loop.
18. `</table>`: Marks the end of the HTML table.
19. `</body>`: Marks the end of the `<body>` section.
20. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that displays the customer dashboard for a shopping website. It shows a personalized greeting to the user, lists the available products in a table with their names, prices, and images. Each product is accompanied by an "Add to Cart" link that allows the user to add the product to their shopping cart.

This is a JSP file that represents the home page of a website for a sporty shoes store. Let's go through the contents of the JSP file:

1. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
2. `<html>`: This tag marks the beginning of the HTML document.
3. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
4. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1.
5. `<title>Sporty Shoes - Welcome to Home Page</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab.
6. `</head>`: Marks the end of the `<head>` section.
7. `<body>`: The `<body>` element contains the visible content of the HTML page.
8. `<h1>Welcome to Sporty Shoes</h1>`: This line displays the heading "Welcome to Sporty Shoes" on the page.
9. `<p>Developed by: Om Shankar Mishra</p>`: This line displays the name of the developer who developed the website.
10. `<p>Your one-stop destination for trendy and comfortable sporty footwear.</p>`: This line provides a brief description of the website's purpose, which is to offer trendy and comfortable sporty footwear.
11. `<a href="./showLogin">Login</a>`: This line creates a hyperlink with the text "Login". The URL of the hyperlink points to "./showLogin", which is likely a URL mapping for the login page of the website.
12. `<a href="./showSignUp">Signup</a>`: This line creates a hyperlink with the text "Signup". The URL of the hyperlink points to "./showSignUp", which is likely a URL mapping for the signup page of the website.

13. `<a href="./showProducts">View Products</a>`: This line creates a hyperlink with the text "View Products". The URL of the hyperlink points to "./showProducts", which is likely a URL mapping for the page that displays the available products in the sporty shoes store.
14. `</body>`: Marks the end of the `<body>` section.
15. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that serves as the home page of the Sporty Shoes website. It welcomes users to the website, provides a brief description of the store's offerings, and offers links for login, signup, and viewing products.

This is a JSP file that represents the login page of the website. Let's go through the contents of the JSP file:

1. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
2. `<html>`: This tag marks the beginning of the HTML document.
3. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
4. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1.
5. `<title>Insert title here</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab. The title is currently set to "Insert title here", but it can be changed to a more meaningful title.
6. `</head>`: Marks the end of the `<head>` section.
7. `<body>`: The `<body>` element contains the visible content of the HTML page.
8. `<form action="./login" method="post">`: This line starts a form that allows users to enter their login credentials. The `action` attribute specifies the URL where the form data will be sent when submitted, and the `method` attribute specifies the HTTP method to be used (in this case, POST).
9. `Username: <input type="text" name="username"><br>`: This line displays the label "Username:" followed by an input field where users can enter their username. The `name` attribute of the input field is set to "username", which will be used to identify the input data on the server side.
10. `Password: <input type="password" name="password"><br>`: This line displays the label "Password:" followed by an input field where users can enter their password. The `type` attribute of the input field is set to "password", which will hide the entered text for security purposes. The `name` attribute is set to "password" to identify the input data on the server side.
11. `<input type="submit" value="Submit">`: This line creates a submit button that users can click to submit the login form.
12. `</form>`: Marks the end of the login form.
13. `<p>${msg}</p>`: This line displays the value of the "msg" attribute, which can be set on the server side to provide messages or feedback to the user.
14. `</body>`: Marks the end of the `<body>` section.
15. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that serves as the login page of the website. It includes a form where users can enter their username and password, and a submit button to send the login credentials to the server. The "msg" attribute can be used to display any messages or feedback from the server, such as login error messages.

This is a JSP file that represents the payment processing page of the website. Let's go through the contents of the JSP file:

1. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
2. `<html>`: This tag marks the beginning of the HTML document.
3. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
4. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1.
5. `<title>Make Payment</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab. The title is set to "Make Payment".
6. `</head>`: Marks the end of the `<head>` section.
7. `<body>`: The `<body>` element contains the visible content of the HTML page.
8. `<h1>Payment Process</h1>`: This line displays the heading "Payment Process" on the page.
9. `<p>Your payment is being processed.</p>`: This line displays a paragraph that informs the user that their payment is being processed.
10. `<p>For learning purposes, this is just a placeholder page.</p>`: This line displays a paragraph explaining that this page is just a placeholder for learning purposes.
11. `<p>In a real application, you would implement the payment gateway integration here.</p>`: This line displays a paragraph indicating that in a real application, the actual payment gateway integration would be implemented here.
12. `<a href="./checkout_success">Back to Home</a>`: This line creates a hyperlink that allows the user to go back to the home page. The `href` attribute is set to "./checkout_success", which will navigate the user to the "checkout_success" page when clicked.
13. `</body>`: Marks the end of the `<body>` section.
14. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that represents the payment processing page of the website. It provides a placeholder message indicating that the payment is being processed and that this page is a placeholder for learning purposes. In a real application, this page would be replaced with the actual payment gateway integration to handle the payment processing. The user is also provided with a link to go back to the home page after the payment processing is completed.

This is a JSP file that represents the payment page of the website. Let's go through the contents of the JSP file:

1. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.

2. `<html>`: This tag marks the beginning of the HTML document.
3. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
4. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1.
5. `<title>Payment</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab. The title is set to "Payment".
6. `</head>`: Marks the end of the `<head>` section.
7. `<body>`: The `<body>` element contains the visible content of the HTML page.
8. `<h1>Payment Page</h1>`: This line displays the heading "Payment Page" on the page.
9. `<form action="/processPayment" method="post">`: This line creates a form element for payment details. The `action` attribute is set to "/processPayment", which specifies the URL where the form data will be sent when the user submits the form. The `method` attribute is set to "post", indicating that the form data will be sent via an HTTP POST request.
10. `Card Number: <input type="text" name="cardNumber"><br>`: This line creates an input field for the card number. The `name` attribute is set to "cardNumber", which will be used as the parameter name when the form is submitted. The user will enter their card number in this field.
11. `<input type="submit" value="Make Payment">`: This line creates a submit button that the user can click to make the payment. When the button is clicked, the form data will be sent to the URL specified in the `action` attribute.
12. `</form>`: Marks the end of the form element.
13. `</body>`: Marks the end of the `<body>` section.
14. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that represents the payment page of the website. It includes a form with an input field for the card number and a submit button. The user will enter their card details and click the "Make Payment" button to initiate the payment process. In a real application, additional input fields for expiration date, CVV, and other payment details would be added to the form. The form data will be sent to the "/processPayment" URL, where the payment processing logic will handle the payment transaction.

This is a JSP file that represents the cart page of the website. Let's go through the contents of the JSP file:

1. `<%@page import="org.simplilearn.entities.CartItem"%>` and `<%@page import="java.util.List"%>`: These lines import the necessary Java classes `CartItem` and `List` to be used in the JSP page.
2. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
3. `<html>`: This tag marks the beginning of the HTML document.
4. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
5. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1.

6. `<title>Cart</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab. The title is set to "Cart".
7. `</head>`: Marks the end of the `<head>` section.
8. `<body>`: The `<body>` element contains the visible content of the HTML page.
9. `<h1>Your Cart</h1>`: This line displays the heading "Your Cart" on the page.
10. `<table border="1">`: This line starts a table with a border.
11. `<tr>`, `<th>`: These tags define the table row and table header cells for the column headings: "Item Name", "Quantity", "Price", and "Action".
12. `<%-- Loop through the cart items and display details --%>`: This is a JSP comment that indicates the start of a loop to display cart items.
13. `<% ... %>`: This is a scriptlet where Java code can be written to access the cart items and display their details.
14. `<%= item.getName() %>`, `<%= item.getQuantity() %>`, `<%= item.getPrice() %>`: These expressions output the name, quantity, and price of each cart item.
15. `<a href="./removeFromCart/<%= item.getPid() %>">Remove</a>`: This line creates a link to remove the cart item. The link's URL includes the cart item's ID (pid) as a parameter.
16. `<%-- End of loop --%>`: This is a JSP comment that indicates the end of the loop for displaying cart items.
17. `</table>`: Marks the end of the table.
18. `<p>Total Cart Value: <%= request.getAttribute("totalCartValue") %></p>`: This line displays the total cart value, which is obtained from the request attributes.
19. `<a href="./checkout">Proceed to Checkout</a>`: This line creates a link to the checkout page, allowing the user to proceed with the checkout process.
20. `</body>`: Marks the end of the `<body>` section.
21. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that represents the cart page of the website. It displays a table with the cart items, including their names, quantities, and prices. Each item has a "Remove" link that allows the user to remove the item from the cart. The total cart value is displayed at the bottom, and the user can proceed to the checkout page using the "Proceed to Checkout" link.

This is a JSP file that represents the purchase report page of the website. Let's go through the contents of the JSP file:

1. `<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>`: This line sets the language and character encoding for the JSP page.
2. `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>`: This line imports the JSTL core tag library, allowing the use of JSTL tags in the page.
3. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
4. `<html>`: This tag marks the beginning of the HTML document.

5. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
6. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1.
7. `<title>Purchase Report</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab. The title is set to "Purchase Report".
8. `</head>`: Marks the end of the `<head>` section.
9. `<body>`: The `<body>` element contains the visible content of the HTML page.
10. `<h1>Purchase Report</h1>`: This line displays the heading "Purchase Report" on the page.
11. `<form method="post" action="/purchaseReport">`: This line starts a form for filtering the purchase report. The form submits the data to the "/purchaseReport" URL using the POST method.
12. `<label for="startDate">Start Date:</label> <input type="date" id="startDate" name="startDate" required>`: This line creates a label and an input field for selecting the start date of the purchase report.
13. `<label for="endDate">End Date:</label> <input type="date" id="endDate" name="endDate" required>`: This line creates a label and an input field for selecting the end date of the purchase report.
14. `<label for="categoryId">Category:</label> <select id="categoryId" name="categoryId" required>`: This line creates a label and a dropdown select element for selecting the category for filtering the purchase report.
15. `<c:forEach var="category" items="${categories}">`: This is a JSTL forEach loop that iterates over the categories available in the model attribute "categories".
16. `<option value="${category.id}">${category.name}</option>`: This line creates an option for each category in the dropdown select. The value of the option is set to the category ID, and the displayed text is the category name.
17. `</c:forEach>`: This ends the forEach loop.
18. `<button type="submit">Filter</button>`: This line creates a submit button to submit the form for filtering the purchase report.
19. `</form>`: Marks the end of the form.
20. `<table>`: This line starts a table for displaying the filtered purchase report.
21. `<tr>`, `<th>`: These tags define the table row and table header cells for the column headings: "Purchase Date", "Category", and "Total Amount".
22. `<c:forEach var="purchase" items="${purchases}">`: This is a JSTL forEach loop that iterates over the purchases available in the model attribute "purchases".
23. `<td>${purchase.purchaseDate}</td>`, `<td>${purchase.category.name}</td>`, `<td>${purchase.totalAmount}</td>`: These expressions output the purchase date, category name, and total amount for each purchase.
24. `</c:forEach>`: This ends the forEach loop.
25. `</table>`: Marks the end of the table.
26. `</body>`: Marks the end of the `<body>` section.
27. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that allows the user to filter the purchase report by selecting a start date, end date, and category. The filtered results are displayed in a table showing the purchase date, category name, and total amount for each purchase that matches the selected criteria.

The provided code is an HTML form for user registration, which is displayed when the "/showSignUp" URL is accessed. Let's break down the form and its elements:

1. `<form>`: This HTML element is used to create an HTML form that will submit user registration data to the "/signUp" URL using the HTTP POST method.
2. `action="./signUp"`: This attribute specifies the URL where the form data will be submitted when the user clicks the "Submit" button. In this case, it's set to "/signUp", which corresponds to the URL mapped in the Spring controller to handle user registration.
3. `method="post"`: This attribute specifies the HTTP method to be used when submitting the form. In this case, it's set to "post", which means the form data will be sent in the request body.
4. `<input type="text" name="username">`: This is an input field for the user to enter their username. The `name` attribute is set to "username", which is used to identify this input field in the request parameter when the form is submitted.
5. `<input type="password" name="password">`: This is an input field for the user to enter their password. The `name` attribute is set to "password", which is used to identify this input field in the request parameter when the form is submitted. Note that the input type is "password," which means the entered text will be masked.
6. `<input type="email" name="email">`: This is an input field for the user to enter their email address. The `name` attribute is set to "email", which is used to identify this input field in the request parameter when the form is submitted. The input type is "email," which can help with form validation and displaying a suitable keyboard on mobile devices.
7. `<select name="role">`: This is a dropdown (select) element for the user to select their role (Customer or Admin). The `name` attribute is set to "role", which is used to identify the selected option in the request parameter when the form is submitted.
8. `<option>`: These are the options within the dropdown. The user can select either "Customer" or "Admin" as their role.
9. `<input type="submit" value="Submit">`: This is the submit button. When the user clicks this button, the form data will be submitted to the specified action URL ("/signUp") using the specified HTTP method (POST).
10. `${msg}`: This is a placeholder for an error message that will be displayed if there is an error during user registration. The error message is set in the model attribute named "msg" in the Spring controller.

Overall, this HTML form allows users to enter their registration details (username, password, email, and role) and submit the form to register as a new user in the application. If there is an error during registration, the error message will be displayed in red below the form.

This is a JSP file that displays a table containing user information. Let's go through the contents of the JSP file:

1. `<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>`: This line sets the language and character encoding for the JSP page.
2. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
3. `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>`: This line imports the JSTL (JavaServer Pages Standard Tag Library) core tag library, which allows us to use JSTL tags in the JSP page.
4. `<html>`: This tag marks the beginning of the HTML document.
5. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
6. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1.
7. `<title>View Users</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab. The title is set to "View Users".
8. `</head>`: Marks the end of the `<head>` section.
9. `<body>`: The `<body>` element contains the visible content of the HTML page.
10. `<h1>View Users</h1>`: This line displays the heading "View Users" on the page.
11. `<table border="1">`: This line starts a table with a border of 1.
12. `<tr>`: This line starts a table row.
13. `<th>Username</th>`: This line creates a table header cell with the text "Username".
14. `<th>Email</th>`: This line creates a table header cell with the text "Email".
15. `<th>Role</th>`: This line creates a table header cell with the text "Role".
16. `</tr>`: This line ends the table header row.
17. `<c:forEach var="user" items="${users}">`: This line uses the JSTL forEach tag to loop through the list of users (retrieved from the "users" attribute in the model) and assigns each user to the variable "user".
18. `<tr>`: This line starts a table row for each user.
19. `<td>${user.username}</td>`: This line creates a table data cell that displays the username of the current user.
20. `<td>${user.email}</td>`: This line creates a table data cell that displays the email of the current user.
21. `<td>${user.role}</td>`: This line creates a table data cell that displays the role of the current user.
22. `</tr>`: This line ends the table row for each user.
23. `</c:forEach>`: This line ends the forEach loop.
24. `</table>`: This line ends the table.
25. `<a href="./adminDashboard">Back to Dashboard</a>`: This line creates a hyperlink that allows the user to navigate back to the admin dashboard.
26. `</body>`: Marks the end of the `<body>` section.
27. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that displays a table of user information. The table includes columns for username, email, and role. It uses JSTL forEach loop to iterate through the list of users and populate the table with their details. The page also provides a link to navigate back to the admin dashboard.

This is a JSP file that displays a table containing product information. Let's go through the contents of the JSP file:

1. `<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>`: This line sets the language and character encoding for the JSP page.
2. `<!DOCTYPE html>`: This line specifies the document type and version of HTML being used.
3. `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>`: This line imports the JSTL (JavaServer Pages Standard Tag Library) core tag library, which allows us to use JSTL tags in the JSP page.
4. `<html>`: This tag marks the beginning of the HTML document.
5. `<head>`: The `<head>` element contains meta-information about the document, such as the character encoding and the page title.
6. `<meta charset="ISO-8859-1">`: This line sets the character encoding of the page to ISO-8859-1.
7. `<title>View Products</title>`: This line sets the title of the page, which will be displayed in the browser's title bar or tab. The title is set to "View Products".
8. `</head>`: Marks the end of the `<head>` section.
9. `<body>`: The `<body>` element contains the visible content of the HTML page.
10. `<h1>View Products</h1>`: This line displays the heading "View Products" on the page.
11. `<table border="1">`: This line starts a table with a border of 1.
12. `<tr>`: This line starts a table row for the table header.
13. `<th>Name</th>`: This line creates a table header cell with the text "Name".
14. `<th>Price</th>`: This line creates a table header cell with the text "Price".
15. `<th>Image</th>`: This line creates a table header cell with the text "Image".
16. `</tr>`: This line ends the table header row.
17. `<c:forEach var="product" items="${products}">`: This line uses the JSTL forEach tag to loop through the list of products (retrieved from the "products" attribute in the model) and assigns each product to the variable "product".
18. `<tr>`: This line starts a table row for each product.
19. `<td>${product.name}</td>`: This line creates a table data cell that displays the name of the current product.
20. `<td>${product.price}</td>`: This line creates a table data cell that displays the price of the current product.
21. `<td><img style="width: 100px; height: 100px" src="${product.imageUrl}" alt="${product.name}" /></td>`: This line creates a table data cell that contains an image tag with the URL of the product's image (retrieved from the "imageUrl" property of the product). The image is displayed with a width and height of 100 pixels and an alt attribute containing the product's name.
22. `</tr>`: This line ends the table row for each product.

23. `</c:forEach>`: This line ends the forEach loop.
24. `</table>`: This line ends the table.
25. `<a href="./">Back to Home</a>`: This line creates a hyperlink that allows the user to navigate back to the home page.
26. `</body>`: Marks the end of the `<body>` section.
27. `</html>`: Marks the end of the HTML document.

Overall, this JSP file generates an HTML page that displays a table of product information. The table includes columns for product name, price, and an image of the product. It uses JSTL forEach loop to iterate through the list of products and populate the table with their details. The page also provides a link to navigate back to the home page.

By: Om Shankar Mishra