Project Report

[ConcordiaEats WebApp]

Group 2 Members: Refat Abuzriba(40161866), Yunqing Chen(40253803), Xinyi Deng (40254082), Yihang Huang(40254370), Yusuke Ishii(40253657), Mengyang Qiu (40253528), Kevin Song Ngy Tea (27035047)

1. Schedule

Objective:

Implement additional admin and user functionality for ConcordiaEats Web App, including admin features for displaying best selling/least selling products and managing sales discounts, and user features such as food item categories and search, personalized recommendations, favorites section, and shopping cart.

Duration: One month

Team: All team members

Timeline:

Week 1

Requirements gathering:

- Consulted with stakeholders (Team members and TA) to establish a stable set of requirements for both admin and user functionalities.
- Determined the necessary features and user interface elements for admin functions (best selling/least selling products and discount management) and user functions (food item categories and search, personalized recommendations, favorites section, and shopping cart).
- Defined the technical requirements and constraints for the implementation of these features.

Feasibility Analysis:

- Analyzed the available resources, such as team members skill sets, project timeline, and technology stack to determine the feasibility of implementing the desired admin and user functionalities.
- Identified potential challenges and limitations that may arise during the implementation process.
- Assessed the feasibility of implementing the features within the given project timeline and with the available resources.

Week 2

Risk Analysis:

- Identified potential risks associated with the implementation of both admin and user functionalities.
- Categorized the risks based on their severity (low, medium, high, critical, catastrophic).
- Assessed the likelihood and impact of each identified risk.

Mitigation Plans:

- Developed mitigation strategies for each identified risk, considering their severity and impact.

- Established contingency plans to handle unexpected issues that may arise during the implementation process.
- Incorporated risk management practices into the project timeline and team assignments.

Week 3

Implementation:

- Developed admin features for displaying best selling/least selling products and managing sales discounts.
- Implemented user features, including food item categories and search, personalized recommendations, favorites section, and shopping cart.
- Ensured that the implemented features are in line with the established requirements and adhere to the project's technical constraints.
- Collaborated with team members to address any challenges or issues that arise during the implementation.

Week 4

Testing Strategies:

- Developed unit tests to validate the functionality and performance of the implemented admin and user features.
- Conducted integration tests to ensure that the new features work seamlessly with the existing system.
- Performed usability tests to verify that user interface and user experience are in line with the project requirements and expectations.

Tools Used:

We used GitHub to manage, keep track of, and merge all files. It also provides time tracking and version control on what functions have been implemented. When implemented, it also reduces the risk of integration difficulties. Finally, it provides tools to facilitate code review.

2. Risk Analysis

The risks that may occur in the project are listed in the table with a mitigation plan. By addressing these risks, we aim to ensure a successful project outcome.

Risk Description	Criticality	Mitigation Plan
Inaccurate project scope and requirements	High	Collaborate with the team and TA to refine project scope, and clarify requirements.
Inadequate time management and scheduling	Medium	Break tasks into smaller, manageable components; assign deadlines and monitor progress regularly.
Insufficient or unclear documentation	Medium	Allocate time for documentation, use templates, and maintain consistency across the project.
Miscommunication among team members	Medium	Hold regular meetings, use collaborative tools, and maintain open communication channels.
Incomplete or incorrect implementation of functionalities	High	Implement code reviews, unit testing, and integration testing to ensure correctness.
Integration issues between components	High	Establish clear interfaces and dependencies between components, and perform integration testing.

3. Materialized Risks

By addressing the materialized risks below, our team was able to overcome implementation challenges and deliver a functional and reliable ConcordiaEats web app:

- Some team members faced challenges implementing certain features due to limited coding experience. To address this issue, we paired experienced members with less experienced ones for knowledge sharing and learning. Additionally, we used inline comments and proper documentation to increase code readability and maintainability.
- Time constraints were a significant concern for our team. To tackle this issue, we prioritized core functionalities and adopted an Agile development approach, which allowed us to deliver a working prototype and iteratively refine the web app based on feedback and testing.
- Integration challenges arose when combining different components of the web app. To mitigate this, we established clear interfaces and dependencies between components and performed integration testing. This approach helped us identify and resolve integration issues early in the development process.
- Although our team had good communication overall, there were occasional misunderstandings about task allocation and progress. To overcome this, we held regular meetings to discuss updates and address any concerns.

- Writing and maintaining unit tests were essential to ensure the reliability of individual components. We utilized parameterized tests to run the same test multiple times with different input sets, reducing test code duplication and increasing testing efficiency.

4. Implementation

Our ConcordiaEats project adopts the Model-View-Controller (MVC) architecture to organize its components and facilitate testing. The structure of the components is as follows:

Model:

Cart.java

● In this class, we import the following libraries:
❖ javax.persistence.Entity: for specifying that the class is an entity and mapped to a database table.
❖ javax.persistence.Table: for defining the table for the entity.
❖ java.io.Serializable: to indicate that instances of this class can be serialized.
❖ java.util.Objects: for comparing object instances and generating hash codes.
❖ javax.persistence.Embeddable: for marking a class as embeddable and allowing it to be used as a composite key.
❖ javax.persistence.EmbeddedId: for specifying a composite primary key.

● The Cart class is an @Entity with a table name "cart".
● It has a composite primary key represented by the CartId class and an Integer field named "quantity".
● The CartId class is marked as @Embeddable and implements Serializable.
● It has two fields, user_id and product_id, both of which are of type Integer.
● Getters and setters are provided for all fields in both the Cart and CartId classes.
● The equals() and hashCode() methods in the CartId class are overridden to compare instances based on the user_id and product_id fields, ensuring uniqueness and proper object comparison.

CartItemInfo.java

● In this class, we import the following library:
❖ com.comp5541.ConcordiaEats.model.Product: for referencing the Product class within this class.

● The CartItemInfo class is a simple Java class representing a cart item with the associated product and its quantity.
● It has two fields: "product" of type Product, and "quantity" of type Integer.
● A constructor is provided to initialize the CartItemInfo instance with the given product and quantity.

Category.java

● In this class, we import the following libraries:
❖ javax.persistence.Entity: for specifying that the class is an entity and mapped to a database table.

❖ javax.persistence.GeneratedValue: for configuring the auto-generated primary key value.
❖ javax.persistence.GenerationType: for specifying the primary key generation strategy.
❖ javax.persistence.Id: for marking the primary key field of the entity.
❖ javax.persistence.Table: for defining the table for the entity.

● The Category class is an @Entity with a table name "categories".
● It has two fields: "categoryid" of type Integer, and "name" of type String.
● The "categoryid" field is marked as the primary key (@Id) and configured to use the GenerationType.IDENTITY strategy for auto-incrementing the primary key value.
● Getters and setters are provided for both the "categoryid" and "name" fields, allowing for easy manipulation of the object properties.

Favorite.java
● In this class, we import the following libraries:
❖ javax.persistence.Entity: for specifying that the class is an entity and mapped to a database table.
❖ javax.persistence.Table: for defining the table for the entity.
❖ java.io.Serializable: to indicate that instances of this class can be serialized.
❖ java.util.Objects: for comparing object instances and generating hash codes.
❖ javax.persistence.Embeddable: for marking a class as embeddable and allowing it to be used as a composite key.
❖ javax.persistence.EmbeddedId: for specifying a composite primary key.

● The Favorite class is an @Entity with a table name "favorites".
● It has a composite primary key represented by the FavoriteId class.
● The FavoriteId class is marked as @Embeddable and implements Serializable.
● It has two fields, user_id and product_id, both of which are of type Integer.
● Getters and setters are provided for all fields in both the Favorite and FavoriteId classes.
● The equals() and hashCode() methods in the FavoriteId class are overridden to compare instances based on the user_id and product_id fields, ensuring uniqueness and proper object comparison.

Product.java
● In this class, we import the following libraries:
❖ javax.persistence.Entity: for specifying that the class is an entity and mapped to a database table.
❖ javax.persistence.GeneratedValue: for configuring the auto-generated primary key value.
❖ javax.persistence.GenerationType: for specifying the primary key generation strategy.
❖ javax.persistence.Id: for marking the primary key field of the entity.
❖ javax.persistence.Table: for defining the table for the entity.

● The Product class is an @Entity with a table name "products".
● It has the following fields:

id (Integer): primary key, auto-incremented
name (String): product name

categoryid (Integer): foreign key reference to the Category entity
image (String): image URL or path
quantity (Integer): available stock
price (Double): product price
weight (Integer): product weight
description (String): product description
onsale (Integer): flag to indicate if the product is on sale
discount (Integer): percentage of discount applied to the product
sold (Integer): number of units sold
● The "id" field is marked as the primary key (@Id) and configured to use the GenerationType.IDENTITY strategy for auto-incrementing the primary key value.
● Getters and setters are provided for all fields, allowing for easy manipulation of the object properties.

User.java
User.java
● In this class, we import the following libraries:
❖ javax.persistence.Entity: for specifying that the class is an entity and mapped to a database table.
❖ javax.persistence.GeneratedValue: for configuring the auto-generated primary key value.
❖ javax.persistence.GenerationType: for specifying the primary key generation strategy.
❖ javax.persistence.Id: for marking the primary key field of the entity.
❖ javax.persistence.Table: for defining the table for the entity.
❖ javax.persistence.Column: for mapping the entity's fields to the columns of the database table.

● The User class is an @Entity with a table name "users".
● It has the following fields:

userId (int): primary key, auto-incremented
username (String): user's username
password (String): user's password
role (String): user's role in the system (e.g., admin, customer)
enabled (Integer): flag to indicate if the user account is active
email (String): user's email address
● The "userId" field is marked as the primary key (@Id) and configured to use the GenerationType.IDENTITY strategy for auto-incrementing the primary key value.
● The @Column annotation is used to map the fields to the corresponding database table columns, specifying the column name.
● Getters and setters are provided for all fields, allowing for easy manipulation of the object properties.

Views
Our ConcordiaEats project utilizes HTML view files to display all of its functionalities.

For the customer dashboard, the HTML view file displays various links to navigate to different pages such as search, categories, favorites, cart, and profile. Each of these pages has its own corresponding HTML view file that displays the relevant information to the user.

Similarly, the admin dashboard also has its own HTML view file that displays links to different functionalities such as categories for insert, update, and delete, discount setting, and product management. The product management section allows the admin to add, delete, update, and display the most-selling and least-selling products.

Controllers

LoginController.java
● Imports: User, UserService, and various Spring MVC annotations.
● The LoginController class is marked as a Spring MVC controller and has session attributes "username" and "user_id".
● It uses an instance of UserService, which is injected using the @Autowired annotation.
● Main methods:

showLoginPage(): Handles GET requests for "/", "/login", and "/admin". Returns the "login" view.
loginUser(): Handles POST requests for "/login". Validates user credentials and redirects to the appropriate page ("/admin/main" for admins, "/main" for customers) while setting session attributes. If invalid, returns an error message and redirects back to the login page.

RegisterController.java
● Imports: UserService and various Spring MVC annotations.
● The RegisterController class is marked as a Spring MVC controller.
● It uses an instance of UserService, which is injected using the @Autowired annotation.
● Main methods:

showRegisterPage(): Handles GET requests for "/register". Returns the "register" view to display the registration page.
registerUser(): Handles POST requests for "/register". Accepts registration form parameters and calls UserService to register a new user. If registration is successful, it redirects back to the registration page with a success message. If unsuccessful, it displays an error message on the registration page.

Admin dashboard

ManagecategoriesController.java
● Imports: CategoryService, various Spring MVC annotations, and java.util.List.
● The ManageCategoriesController class is marked as a Spring MVC controller.
● It uses an instance of CategoryService, which is injected using the @Autowired annotation.
● Main methods:

deleteCategories(): Handles POST requests for "/admin/deleteCategories". Accepts category ID as a parameter, and calls CategoryService to delete the category with the specified ID. Redirects to "/admin/categories" upon completion.
updateCategories(): Handles POST requests for "/admin/updateCategories". Accepts category ID and new category name as parameters, and calls CategoryService to update the category. Redirects to "/admin/categories" upon completion.
insertCategories(): Handles POST requests for "/admin/insertCategories". Accepts a new category name as a parameter, and calls CategoryService to insert a new category. Redirects to "/admin/categories" upon completion.

ManageCustomerController.java
● Imports: UserService, various Spring MVC annotations, and java.util.List.
● The ManageCustomersController class is marked as a Spring MVC controller.
● It uses an instance of UserService, which is injected using the @Autowired annotation.
● Main method:

deleteUsers(): Handles POST requests for "/admin/deleteUsers". Accepts a username as a parameter, and calls UserService to delete the user with the specified username. Redirects to "/admin/customers" upon completion.

ManageDiscountController.java
● Imports: ProductService, various Spring MVC annotations.
● The ManageDiscountController class is marked as a Spring MVC controller and contains a session attribute "username".
● It uses an instance of ProductService, which is injected using the @Autowired annotation.
● Main methods:

resetDiscount(): Handles POST requests for "/admin/resetDiscount". Takes an Integer 'id' as a request parameter, resets the discount for the product with the given 'id' using ProductService, and redirects to the "/admin/discounts" URL.
updateDiscount(): Handles POST requests for "/admin/applyDiscount". Takes an Integer 'discount' and an Integer 'id' as request parameters, updates the discount for the product with the given 'id' using ProductService, and redirects to the "/admin/discounts" URL.

ManageProductController.java
● Imports: ProductService, Product model, various Spring MVC annotations.
● The ManageProductsController class is marked as a Spring MVC controller and contains a session attribute "username".
● It uses an instance of ProductService, which is injected using the @Autowired annotation.
● Main methods:

deleteProducts(): Handles POST requests for "/admin/deleteProducts". Takes an Integer 'id' as a request parameter, deletes the product with the given 'id' using ProductService, and redirects to the "/admin/products" URL.

updateProducts(): Handles POST requests for "/admin/updateProducts". Takes several parameters including 'id', 'newname', 'categoryid', 'quantity', 'price', 'weight', 'description', and 'image'. Sets a default image if not provided. Utilizes a map to store category names, and updates the product using ProductService. Displays success/error messages based on the update outcome. Retrieves a list of products and returns to the "adminProducts" view.

insertProducts(): Handles POST requests for "/admin/insertProducts". Takes several parameters including 'name', 'categoryid', 'quantity', 'price', 'weight', 'description', and 'image'. Sets a default image if not provided. Utilizes a map to store category names, and inserts a new product using ProductService. Displays success/error messages based on the insert outcome. Retrieves a list of products and returns to the "adminProducts" or "adminInsertProducts" view depending on the outcome.

AdminSellingController.java
- Imports: ProductService, Product model, and various Spring MVC annotations.
- The AdminSellingController class is marked as a Spring MVC controller.
- It uses an instance of ProductService, which is injected using the @Autowired annotation.
- Main method:

showAdminProductsPage(): Handles GET requests for "/admin/selling". Creates a map with category names, and adds it to the model. Calls the 'searchSelling()' method from ProductService to get a list of products that are selling, adds the list to the model, and returns the "adminSelling" view.

Customer dashboard

CustomerCategoriesController.java
- Imports: Spring MVC annotations.
- The CustomerCategoriesController class is marked as a Spring MVC controller.
- Main method:

showCustomerCategoriesPage(): Handles GET requests for "/categories". This method can be used to add any additional data to the model if needed, such as a list of categories to display on the page. Currently, no data is added to the model. The method returns the "customerCategories" view.

SearchController.java
- Imports: Spring MVC annotations, Java utility classes, and ConcordiaEats model, repository, and service classes.
- The SearchController class is marked as a Spring MVC controller with session attributes "username" and "user_id".

● Dependencies: ProductService, CartService, FavoriteRepository, and CartRepository.
● Main methods:

showSearchPage(): Handles GET requests for "/search". It queries the database for product IDs added to favorites and cart, adds them to the model, and provides category names as well. Calls the searchProducts() method to retrieve all products and adds them to the model. Returns the "search" view.
addToFavorites(): Handles POST requests for "/addToFavorites". Adds a product to the user's favorites by calling the addProductToFavorites() method. Redirects to the search page with a success or error message.
addToCart(): Handles POST requests for "/addToCart". Adds a product to the user's cart with the specified quantity by calling the addProductToCart() method. Redirects to the search page with a success or error message.

FavoriteController.java
● Imports: Spring MVC annotations, Java utility classes, and ConcordiaEats model, service, and repository classes.
● The FavoriteController class is marked as a Spring MVC controller with a session attribute "user_id".
● Dependencies: FavoriteService, CartService, and CartRepository.
● Main methods:

showFavoritesPage(): Handles GET requests for "/favorites". It retrieves product IDs added to the cart and adds them to the model, along with category names. It calls the getFavoriteProductsByUserId() method to get the user's favorite products and adds them to the model. Returns the "favorites" view.
removeFromFavorites(): Handles POST requests for "/removeFromFavorites". Removes a product from the user's favorites by calling the removeProductFromFavorites() method. Redirects to the favorites page with a success or error message.
addToCart(): Handles POST requests for "/addToCartF". Adds a product to the user's cart with the specified quantity by calling the addProductToCart() method. Redirects to the favorites page with a success or error message.

CartController.java
● Imports: Spring MVC annotations, Java utility classes, and ConcordiaEats model and service classes.
● The CartController class is marked as a Spring MVC controller with a session attribute "user_id".
● Dependencies: CartService.
● Main methods:

showCartPage(): Handles GET requests for "/cart". It defines and adds the category names mapping to the model, then retrieves the list of cart items for the current user and adds them to the model. Returns the "cart" view.
updateCartQuantity(): Handles POST requests for "/updateCartQuantity". It updates the cart quantity for the specified user and product by calling the updateCartQuantity() method. Returns a success response if the update is successful or an error response if an exception occurs.
removeFromCart(): Handles POST requests for "/removeFromCart". It removes a product from the user's cart by calling the removeFromCart() method. Redirects to the cart page with a success or error message.

5. Testing Strategies

The testing section of our project encompasses a comprehensive suite of unit tests designed to validate the functionality and reliability of both the model and controller components within our application. Utilizing the JUnit testing framework, our development team conducted rigorous tests to ensure the correctness of various methods, behaviors, and interactions. These tests enabled us to identify and address any potential issues early in the development process, thereby enhancing the overall quality of the software.

Model Test

UserTest Summary

- Class: UserTest in package com.comp5541.ConcordiaEats.model
- Purpose: Test getter and setter methods of User class
    - Key Test Cases:
    - Verify setId and getId methods for user ID
    - Verify setUsername and getUsername methods for username
    - Verify setPassword and getPassword methods for password
    - Verify setRole and getRole methods for user role
    - Verify setEnabled and getEnabled methods for enabled status
    - Verify setEmail and getEmail methods for email address

ProductTest Summary

- Class: ProductTest in package com.comp5541.ConcordiaEats.model
- Purpose: Test getter and setter methods of Product class
    - Key Test Cases:
    - Verify setId and getId methods for product ID
    - Verify setName and getName methods for product name
    - Verify setCategoryid and getCategoryid methods for category ID
    - Verify setImage and getImage methods for image URL
    - Verify setQuantity and getQuantity methods for quantity
    - Verify setPrice and getPrice methods for price
    - Verify setOnsale and getOnsale methods for onsale status
    - Verify setDiscount and getDiscount methods for discount
    - Verify setWeight and getWeight methods for weight
    - Verify setDescription and getDescription methods for description
    - Verify setSold and getSold methods for sold count

CategoryTest Summary

- Class: CategoryTest in package com.comp5541.ConcordiaEats.model
- Purpose: Test behavior of Category class

- Key Test Case:
    - testCategory: Verify properties of two distinct Category objects
    - Create category1 with categoryid set to 1 and name set to "Electronics"
    - Assert that category1 properties are set correctly
    - Create category2 with categoryid set to 2 and name set to "Books"
    - Assert that category2 properties are set correctly
    - Assert that category1 and category2 are not equal

FavoriteIdTest Summary

- Class: FavoriteIdTest in package com.comp5541.ConcordiaEats.model
- Purpose: Test behavior of Favorite.FavoriteId class
    - Key Test Case:
        - testEqualsAndHashCode: Verify equality and hash code behavior of FavoriteId objects
        - Create favoriteId1 with user_id set to 1 and product_id set to 100
        - Create favoriteId2 with the same user_id and product_id as favoriteId1
        - Create favoriteId3 with user_id set to 2 and product_id set to 200
        - Assert that favoriteId1 and favoriteId2 are equal and have equal hash codes
        - Assert that favoriteId1 and favoriteId3 are not equal and have different hash codes

CartIdTest Summary

- Class: CartIdTest in package com.comp5541.ConcordiaEats.model
- Purpose: Test behavior of Cart.CartId class
    - Key Test Case:
        - testEqualsAndHashCode: Verify equality and hash code behavior of CartId objects
        - Create cartId1 with user_id set to 1 and product_id set to 100
        - Create cartId2 with the same user_id and product_id as cartId1
        - Create cartId3 with user_id set to 2 and product_id set to 200
        - Assert that cartId1 and cartId2 are equal and have equal hash codes
        - Assert that cartId1 and cartId3 are not equal and have different hash codes

Controller Test

LoginControllerTest Summary

- Class: LoginControllerTest in package com.comp5541.ConcordiaEats.controller
- Purpose: Test behavior of LoginController class
    - Key Test Cases:
        - testSuccessfulLoginAsAdmin: Verify successful login as admin user
        - Mock User object representing admin user with username "admin", password "admin123", and role "ROLE_ADMIN"
        - Mock userService to return admin user when provided valid credentials

        - Perform POST request to "/login" with valid admin credentials
        - Expect redirection to "/admin/main"
        - testSuccessfulLoginAsCustomer: Verify successful login as customer user
        - Mock User object representing customer user with username "customer", password "cust123", and role "ROLE_CUSTOMER"
        - Mock userService to return customer user when provided valid credentials
        - Perform POST request to "/login" with valid customer credentials
        - Expect redirection to "/main"
        - testFailedLogin: Verify failed login attempt
        - Mock userService to return null when provided invalid credentials
        - Perform POST request to "/login" with invalid credentials
        - Expect view name "login" with error message "Invalid username or password."

RegisterControllerTest Summary

- Class: RegisterControllerTest in package com.comp5541.ConcordiaEats.controller
- Purpose: Test behavior of RegisterController class
    - Key Test Cases:
        - testShowRegisterPage: Verify display of register page
        - Perform GET request to "/register"
        - Expect view name "register" with status OK (200)
        - testRegisterUser_Success: Verify successful user registration
        - Mock userService to return success message for valid registration data
        - Perform POST request to "/register" with valid registration data
        - Expect view name "register" with status OK (200)
        - Expect model attribute "isSuccess" set to true
        - testRegisterUser_Failure: Verify failed user registration due to password mismatch
        - Mock userService to return error message for mismatching passwords
        - Perform POST request to "/register" with mismatching passwords
        - Expect view name "register" with status OK (200)
        - Expect model attribute "errorMessage" set to "Error: Passwords do not match."

SearchControllerTest Summary

- Class: SearchControllerTest in package com.comp5541.ConcordiaEats.controller
- Purpose: Test behavior of SearchController class
    - Key Test Cases:
        - testAddToFavorites: Verify adding a product to favorites
        - Perform POST request to "/addToFavorites" with a user ID and product ID in session and request parameters
        - Expect redirection to "/search" with status 3xx (redirection)
        - Expect no error message in the model
        - testAddToCart: Verify adding a product to the shopping cart

        - Perform POST request to "/addToCart" with a user ID, product ID, and quantity in session and request parameters
        - Expect redirection to "/search" with status 3xx (redirection)
        - Expect no error message in the model
    - Dependencies Mocked:
        - ProductService
        - CartService
        - FavoriteRepository
        - CartRepository

FavoriteControllerTest Summary

- Class: FavoriteControllerTest in package com.comp5541.ConcordiaEats.controller
- Purpose: Test behavior of FavoriteController class
    - Key Test Cases:
        - testRemoveFromFavorites: Verify removing a product from favorites
        - Perform POST request to "/removeFromFavorites" with a user ID and product ID in session and request parameters
        - Expect redirection to "/favorites" with status 3xx (redirection)
        - Expect success message "Product removed from favorites successfully." in flash attributes
        - testAddToCartF: Verify adding a product from favorites to the shopping cart
        - Perform POST request to "/addToCartF" with a user ID, product ID, and quantity in session and request parameters
        - Expect redirection to "/favorites" with status 3xx (redirection)
        - Expect success message "Product added to cart successfully." in flash attributes
    - Dependencies Mocked:
        - FavoriteService
        - CartService

CartControllerTest Summary

- Class: CartControllerTest in package com.comp5541.ConcordiaEats.controller
- Purpose: Test behavior of CartController class
    - Key Test Cases:
        - testUpdateCartQuantity: Verify updating the quantity of a product in the cart
        - Mock CartService to do nothing when updateCartQuantity method is called
        - Invoke updateCartQuantity method on CartController with sample data
        - Expect response status code 200 and success message "Cart quantity updated successfully."
        - testRemoveFromCart: Verify removing a product from the cart
        - Mock RedirectAttributes object
        - Invoke removeFromCart method on CartController with sample data
        - Expect redirect URL to be "/cart"

        - testPerformCheckout: Verify performing checkout on the cart
        - Mock Model object
        - Mock CartService to return a list of CartItemInfo when getCartItemsByUserId is called
        - Invoke performCheckout method on CartController with sample data
        - Expect view name to be "cart"
    - Dependencies Mocked:
        - CartService
        - CheckoutService

ManageProductsControllerTest Summary

- Class: ManageProductsControllerTest in package com.comp5541.ConcordiaEats.controller
- Purpose: Test behavior of ManageProductsController class
    - Key Test Cases:
        - testDeleteProducts: Verify deleting a product
        - Mock ProductService to verify deletion
        - Invoke deleteProducts method on ManageProductsController with sample product ID
        - Verify deleteProduct method is called on ProductService with product ID
        - Expect redirect URL to be "/admin/products"
        - testUpdateProducts: Verify updating product information
        - Mock ProductService to return a list of products and verify update
        - Invoke updateProducts method on ManageProductsController with sample data
        - Verify updateProduct method is called on ProductService with updated product data
        - Expect view name to be "adminProducts"
        - testInsertProducts: Verify inserting a new product
        - Mock ProductService to return a list of products and verify insertion
        - Invoke insertProducts method on ManageProductsController with sample data
        - Verify insertProduct method is called on ProductService with new product data
        - Expect view name to be "adminProducts"
    - Dependencies Mocked:
        - ProductService
        - Model

ManageDiscountControllerTest Summary

- Class: ManageDiscountControllerTest in package com.comp5541.ConcordiaEats.controller
- Purpose: Test behavior of ManageDiscountController class
    - Key Test Cases:
        - testResetDiscount: Verify resetting the discount for a product
        - Define a product ID for which to reset the discount

        - Invoke resetDiscount method on ManageDiscountController with product ID
        - Verify resetDiscount method is called on ProductService with product ID
        - Expect redirect URL to be "/admin/discounts"
        - testUpdateDiscount: Verify updating the discount for a product
        - Define a product ID and discount value for updating discount
        - Invoke updateDiscount method on ManageDiscountController with discount and product ID
        - Verify updateDiscount method is called on ProductService with discount and product ID
        - Expect redirect URL to be "/admin/discounts"
    - Dependencies Mocked:
        - ProductService

6. Quality Control

During the quality control assessment of our project, it was identified that the base code recommended in Assignment 1 (https://github.com/jaygajera17/E-commerce-project-springBoot) did not fully adhere to the Model-View-Controller (MVC) architectural pattern. Specifically, front-end and back-end components were intertwined, and there was a lack of clear separation among the controllers. In response to these findings, a comprehensive restructuring was undertaken to bring the project into compliance with the MVC pattern. The implementation of MVC principles has resulted in enhanced modularity and improved organization within the codebase. As a result, the project now demonstrates greater maintainability and robustness, which aligns with our quality control objectives.

Building upon the restructuring efforts to align with the Model-View-Controller (MVC) architectural pattern, the ConcordiaEats web application utilizes the Spring MVC framework to effectively organize its components. The application's architecture is designed with five key layers: Model, Repository, Service, Controller, and View. Each layer serves a specific purpose and contributes to achieving a clean separation of concerns. A detailed breakdown of these layers within the context of ConcordiaEats is as follows:

- Model: This layer encapsulates the data and business logic of the application. It is composed of classes representing entities such as User, Product, Category, Cart, and Favorite. These classes, often annotated with @Entity, define the application's data structures, relationships, and properties for database persistence.
- Repository: The Repository layer facilitates interactions with the database by providing interfaces with methods for data querying and manipulation. Extending Spring Data JPA CrudRepository or JpaRepository interfaces, this layer enables basic CRUD operations. Notable repositories within ConcordiaEats include UserRepository, ProductRepository, and others.
- Service: Responsible for implementing business logic, the Service layer processes requests, performs computations, and interacts with the Model and Repository layers. Annotated with @Service, classes within this layer contain methods for specific business operations, such as those found in UserService, ProductService, and other services.

- Controller: Serving as the bridge between the View and Model, the Controller layer manages incoming HTTP requests and returns appropriate responses. Annotated with @Controller, classes within this layer define methods mapped to URL patterns using @GetMapping and @PostMapping. Controllers in ConcordiaEats include CustomerController, AdminController, and more.
- View: Representing the user interface (UI), the View layer defines the appearance of the application's pages using HTML templates created with the Thymeleaf templating engine. Examples of views include pages for the main interface, product listing, user login, and admin dashboard.

In summary, the implementation of the Spring MVC architecture in ConcordiaEats has yielded clear and modular separation among the application's components, thereby enhancing maintainability and scalability.

7. Revision History and Contributions

7.1. Version Details

Version	1.0
Completed on	2023/04/14

7.2. Contribution Log

Section	Contributor	Time Spent (Hrs)	Date
User	Yunqing, Xinyi, Yihang, Mengyang	28	2023/04/11 2023/04/12 2023/04/13
Admin	Yuskue, Kevin, Refat	18	2023/04/12 2023/04/14
MVC Restructuring	Mengyang, Yihang	20	2023/04/13 2023/04/14
Report Writing	The whole team	5	2023/04/14