VIETNAM NATIONAL UNIVERSITY – HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

WEB APPLICATION DEVELOPMENT
IT093IU

FINAL PROJECT REPORT

Topic: Coffee Management System

1.	Huỳnh Tuấn Anh	ITITIU23003
2.	Trần Thị Trúc Mai	ITCSIU23024

Instructor:
Assoc. Prof. Nguyen Van Sinh
Msc. Nguyen Trung Nghia

TABLE OF CONTENT

CHAPTER 1: INTRODUCTION

1. Abstract

- This project demonstrates how we can apply technology, techniques, and theoretical knowledge in the Web Application Development course to design and implement a coffee management system web-based application. By featuring a user-friendly interface, this website helped the to browse menus, customize drinks, place orders, and purchase beverages and cakes conveniently for home delivery.
- The project integrates core concepts such as system analysis and design, database management, server-side programming, and client-side interaction to deliver a complete end-to-end solution. In addition, essential algorithms for order processing, cart management, and price calculation are implemented to ensure accuracy and efficiency. The overall development process, including system architecture, functional modules, and implementation techniques, is explained in detail to demonstrate how theoretical knowledge can be transformed into a practical, real-world web application.

2. Objectives

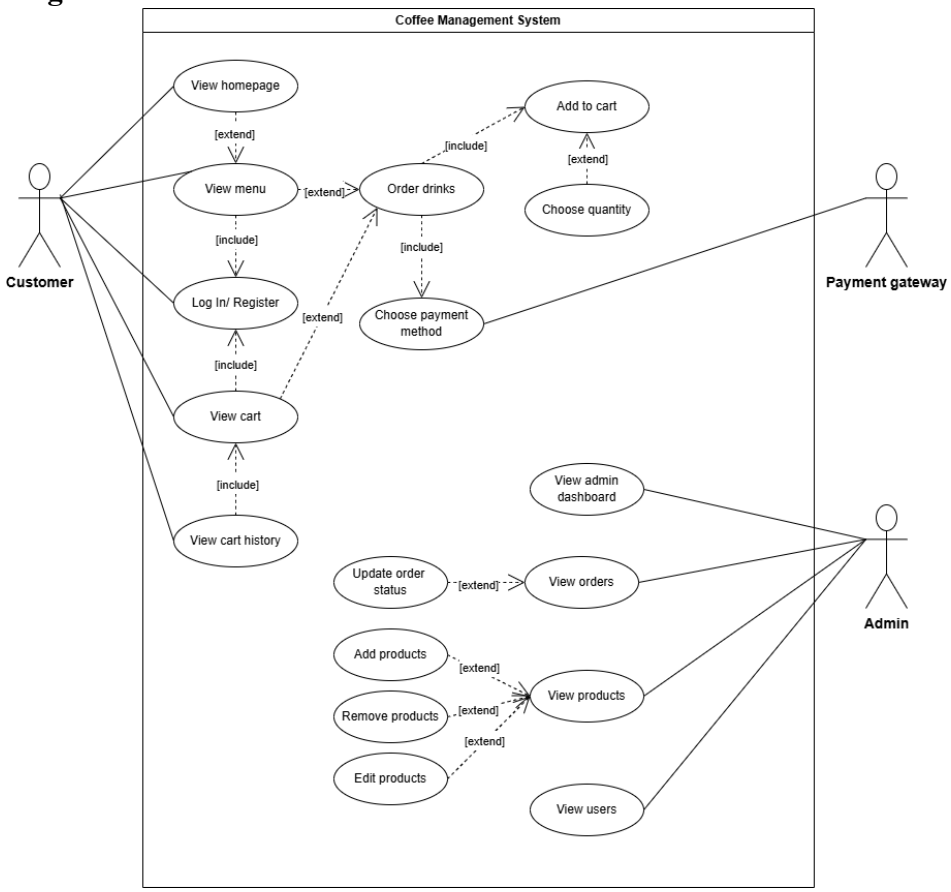
- To apply the techniques, tools, and knowledge learned in the Web Application Development course to a real-world application scenario.
- To design and develop a functional coffee management system that supports online ordering and product customization.
- To build a user-friendly web interface that enhances user experience and simplifies the ordering process.
- To implement core system functionalities such as user authentication, cart management, order processing, and payment handling.
- To strengthen practical skills in web technologies, system design, and problem-solving through hands-on development experience.

3. Tools and programming languages

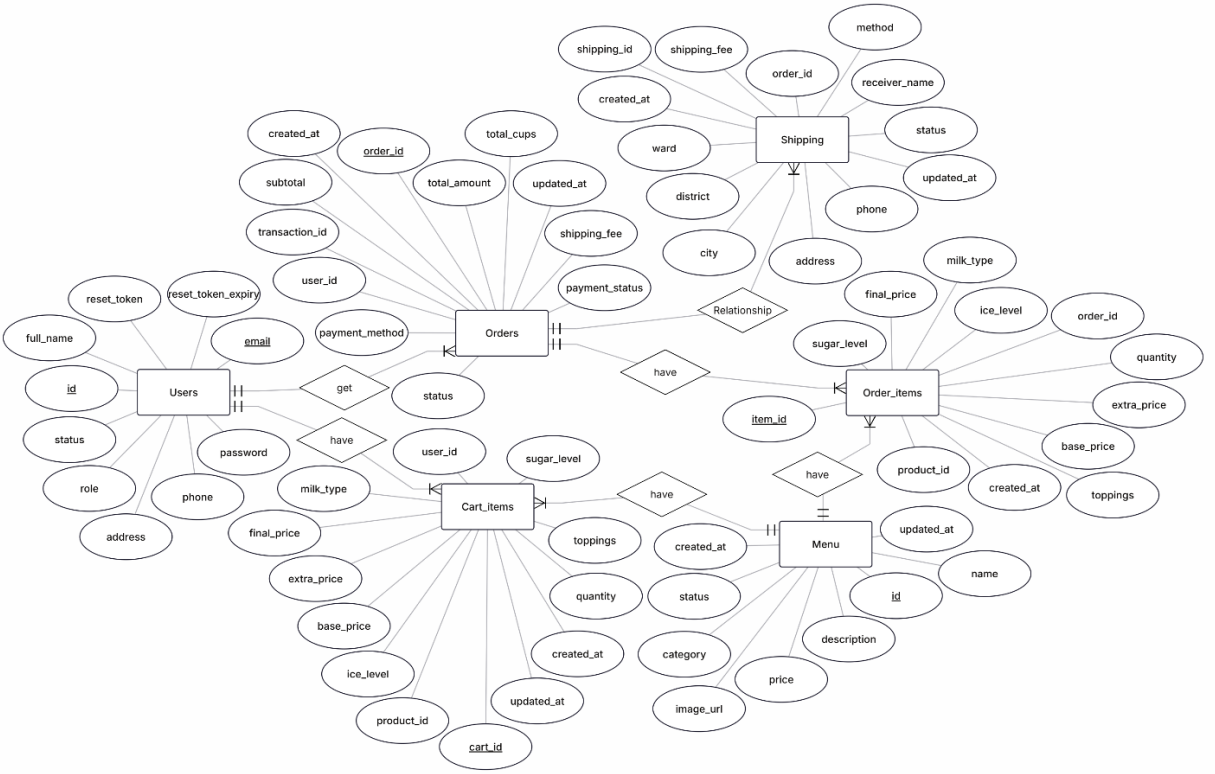
- Java: The primary programming language used for developing the coffee management system, responsible for handling both business logic and server-side processing.
+ Front-end side: Java Server Page (JSP): Used to generate dynamic web content and manage the presentation layer of the application.
+ Back-end side: Java Servlet: Employed to handle server-side logic, process user requests, manage sessions, and interact with the database
- Cascading Style Sheet (CSS): Designing the website.
- JavaScript: Applied on the client side to handle interactive features, implement front-end algorithms, manage real-time countdown functionality, and integrate external APIs.
- Visual Studio Code/Eclipse: Editor/IDE for implementation and testing of the website.
- Git and GitHub: Used to manage source code versions, track changes, and store the project repository collaboratively.
- MySQL: Serves as the primary database system for storing and managing user information, product data, shopping carts, orders, and shipping details.

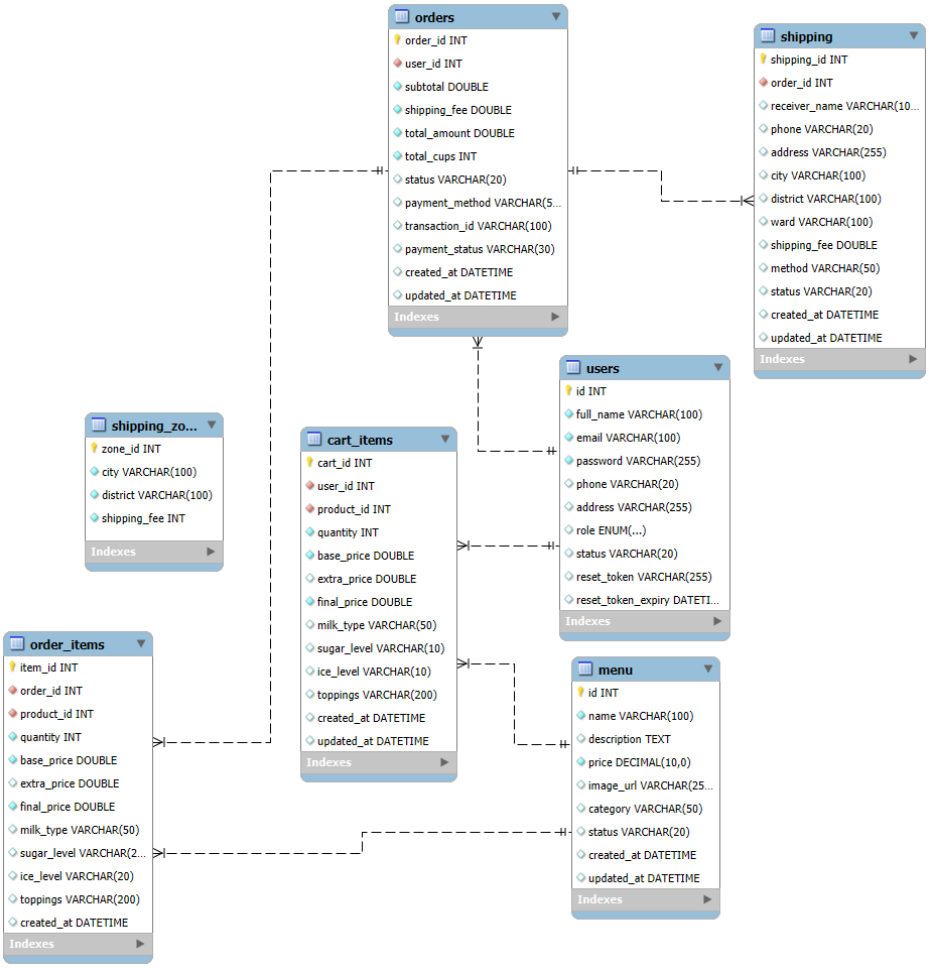
CHAPTER 2: SYSTEM DESIGNS

1. Use case diagram



2. Entity-relationships Diagram (ERD)

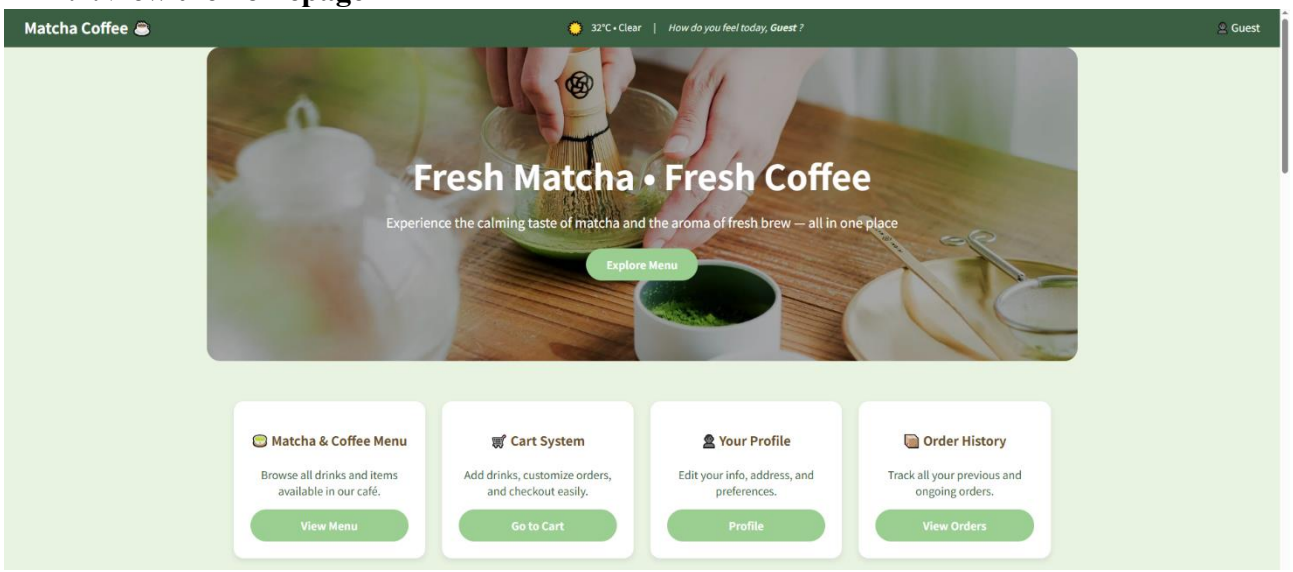



orders
order_id INT
user_id INT
subtotal DOUBLE
shipping_fee DOUBLE
total_amount DOUBLE
total_cups INT
status VARCHAR(20)
payment_method VARCHAR(5...
transaction_id VARCHAR(100)
payment_status VARCHAR(30)
created_at DATETIME
updated_at DATETIME
Indexes
shipping
shipping_id INT
order_id INT
receiver_name VARCHAR(10...
phone VARCHAR(20)
address VARCHAR(255)
city VARCHAR(100)
district VARCHAR(100)
ward VARCHAR(100)
shipping_fee DOUBLE
method VARCHAR(50)
status VARCHAR(20)
created_at DATETIME
updated_at DATETIME
Indexes
users
id INT
full_name VARCHAR(100)
email VARCHAR(100)
password VARCHAR(255)
phone VARCHAR(20)
address VARCHAR(255)
role ENUM(...)
status VARCHAR(20)
reset_token VARCHAR(255)
reset_token_expiry DATETI...
Indexes
shipping_zo...
zone_id INT
city VARCHAR(100)
district VARCHAR(100)
shipping_fee INT
Indexes
cart_items
cart_id INT
user_id INT
product_id INT
quantity INT
base_price DOUBLE
extra_price DOUBLE
final_price DOUBLE
milk_type VARCHAR(50)
sugar_level VARCHAR(10)
ice_level VARCHAR(10)
toppings VARCHAR(200)
created_at DATETIME
updated_at DATETIME
Indexes
order_items
item_id INT
order_id INT
product_id INT
quantity INT
base_price DOUBLE
extra_price DOUBLE
final_price DOUBLE
milk_type VARCHAR(50)
sugar_level VARCHAR(2...
ice_level VARCHAR(20)
toppings VARCHAR(200)
created_at DATETIME
Indexes
menu
id INT
name VARCHAR(100)
description TEXT
price DECIMAL(10,0)
image_url VARCHAR(25...
category VARCHAR(50)
status VARCHAR(20)
created_at DATETIME
updated_at DATETIME
Indexes

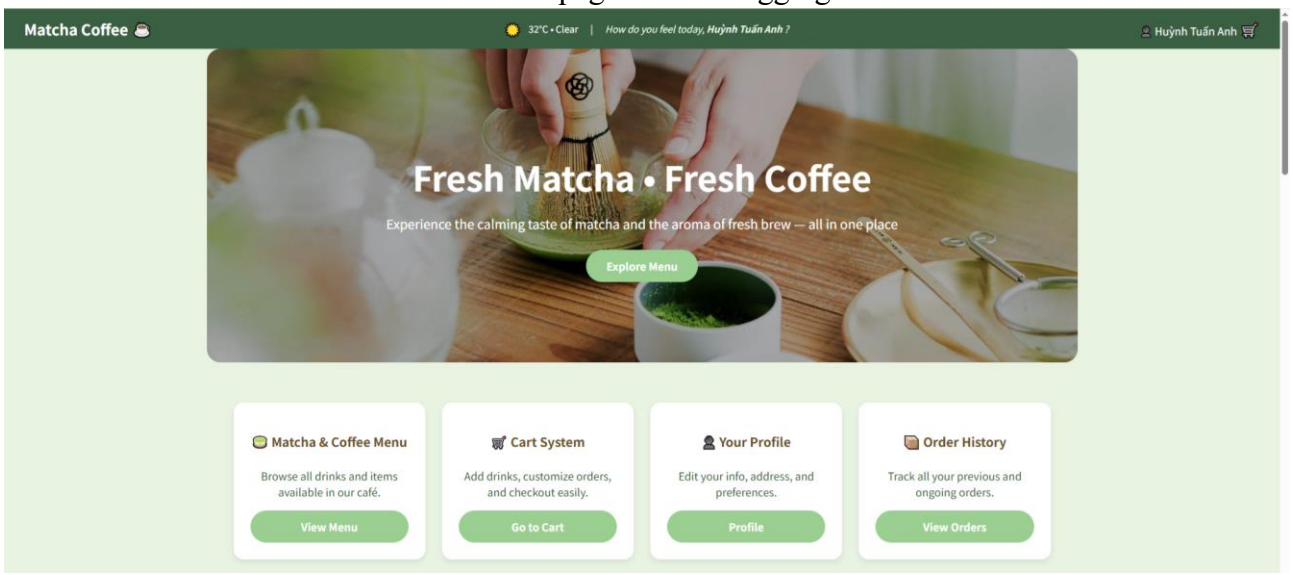
CHAPTER 3: IMPLEMENTATION

1. User's account permission

1.1.View the homepage

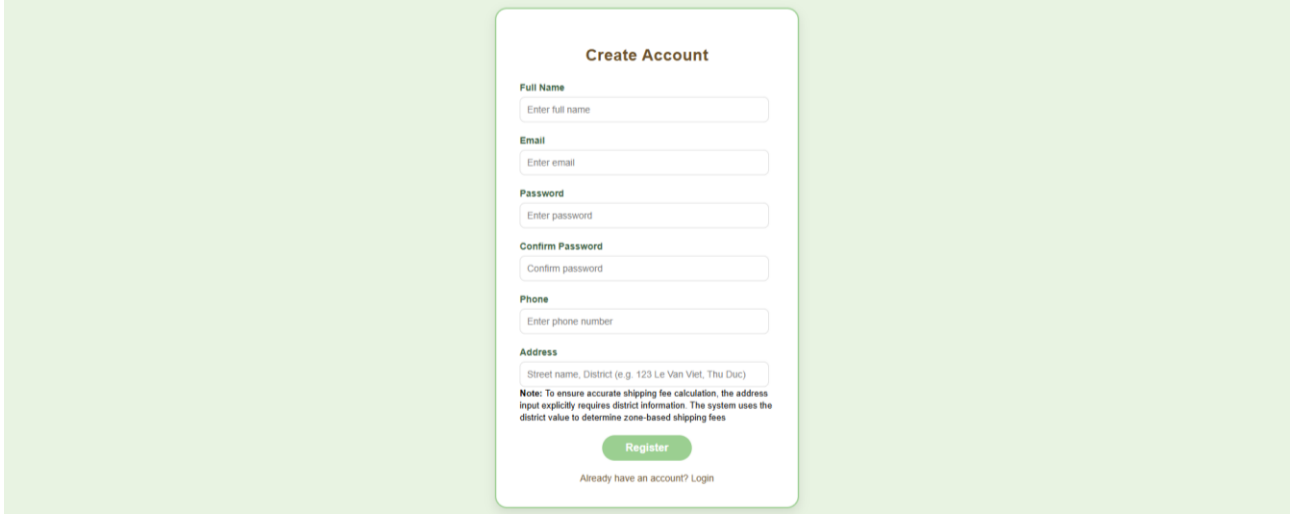


The homepage without logging in



The homepage with logging in

1.2.Registration



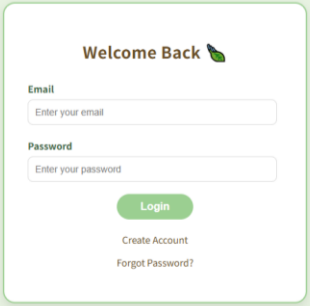
The page of registration (new account)

Explanation of the code flow of the registration:

- Browser → GET /register
+ The user clicks **Create Account** or navigates to the register page.
+ A GET request is sent to /register.
- RegisterController receives GET request: The doGet() method in RegisterController is executed.
- Controller forwards request to register.jsp
+ The registration form is displayed.
+ Any previous error or success messages are shown to the user.
- User fills in registration information: Full name, email, password, confirm password, phone number, address including district for shipping calculations).
- Browser → POST /register
+ The user submits the registration form.
+ A POST request containing all form fields is sent to the server.
- RegisterController receives POST request: The doPost() method retrieves all parameters from the request.
- Controller performs basic validation:
+ The controller checks required fields (full name, email, password, confirm password).
+ If any required field is empty: An error message is set and the request is forwarded back to register.jsp.
- Controller validates email format:
+ The controller checks the email against a regex pattern.
+ If the email format is invalid: An error message is attached to the request, and the user remains on the registration page.
- Controller checks password confirmation:
+ The controller verifies that password and confirmPassword match.
+ If they do not match: An error message is displayed, and the request is forwarded back to register.jsp.
- Controller checks email duplication:
+ The controller calls UserDAO.getUserByEmail(email).
+ The DAO queries the database to check if the email already exists.
+ If the email is already registered: An error message is shown, and the user is redirected back to the registration page.
- Controller hashes the password:
+ The raw password is encrypted using PasswordHashUtil (bcrypt).
+ The system never stores plain-text passwords.
- Controller creates User object: User information is populated by full name, email, hashed password, phone, address, default role is customer, and default status is active.
- Controller calls UserDAO.registerUser(user):
+ The controller delegates database insertion to the DAO layer.
+ The controller does not directly execute SQL.
- UserDAO executes SQL INSERT:
+ The DAO creates a prepared statement.
+ User data is inserted into the users table.
+ SQL injection is prevented using prepared statements.
- DAO returns registration result:
+ If insertion succeeds → returns true.

+ If insertion fails → returns false.
– Controller handles registration success:
+ A success message is attached to the request.
+ The user is forwarded to login.jsp.
+ The user is instructed to log in with the new account.
– Controller handles registration failure: If registration fails:
+ An error message is displayed.
+ The user remains on register.jsp.
– HTML response sent to browser: The final page (login or register) is rendered and returned to the browser.

1.3.Login

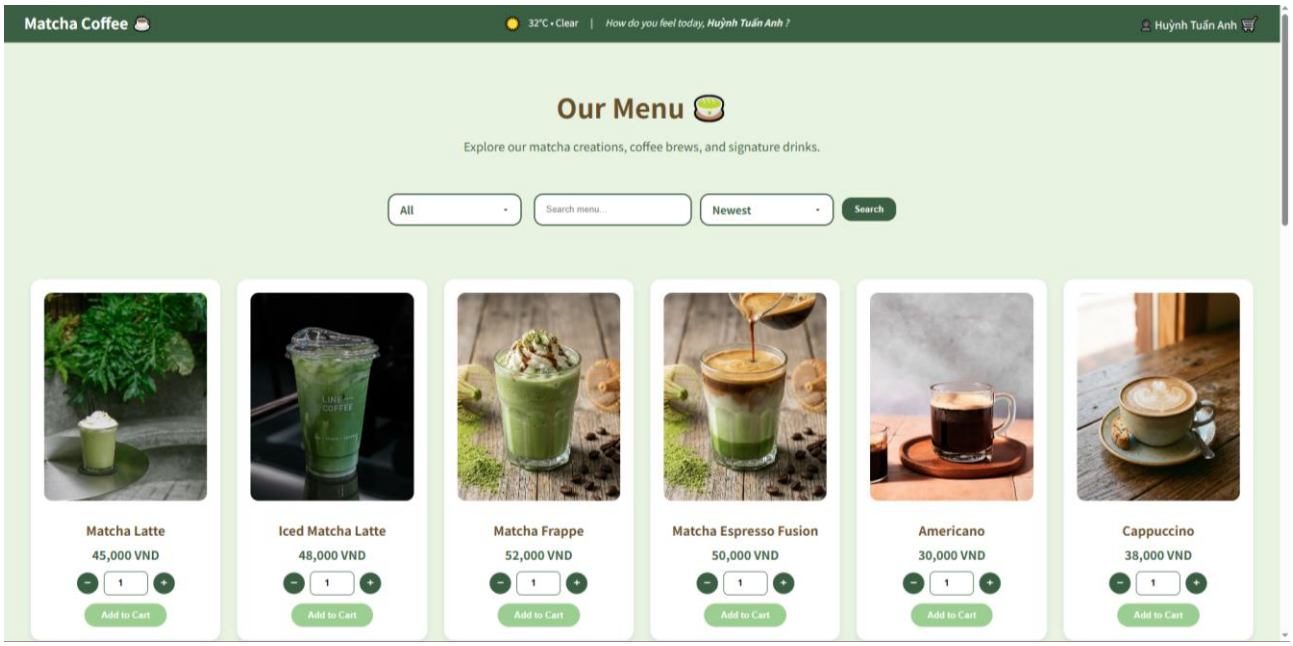


Explanation of the flow of logging in (with authentication):

– Browser → GET /login
+ The user clicks the Login button or accesses the login page directly.
+ The browser sends a GET request to /login.
+ The request is handled by LoginController.java.
– Server (LoginController.java) handles GET request:
+ The doGet() method in LoginController is executed.
+ The controller forwards the request to login.jsp.
+ The login page is displayed, allowing the user to enter email and password.
+ Any error or success message from previous actions is shown using JSTL tags.
– Filter checks for static resources
+ The filter first checks if the request is for static files such as: CSS files, JavaScript files, images, fonts.
+ If the path matches these resources: The request is immediately allowed using chain.doFilter().
+ This prevents authentication checks from blocking website styling or scripts.
– Filter checks for public URLs
+ The filter compares the requested path with a predefined list of public paths, such as: Login page, register page, forgot password, homepage.
+ These URLs do not require authentication.
+ If the requested path is public: The request is allowed to proceed.
– Filter checks authentication status
+ If the URL is not static and not public, the filter treats it as a protected resource.
+ The filter retrieves the current HTTP session (without creating a new one).
+ It checks whether userId exists in the session.
– User is not authenticated
+ If the session does not exist or userId is null: The user is considered not logged in and the filter redirects the browser to login.jsp.

+ The requested protected resource is not accessed.
– User is authenticated
+ If userId exists in the session: The user is considered authenticated and the filter allows the request to continue.
+ The request proceeds to the target servlet or JSP.

1.4.View the menu

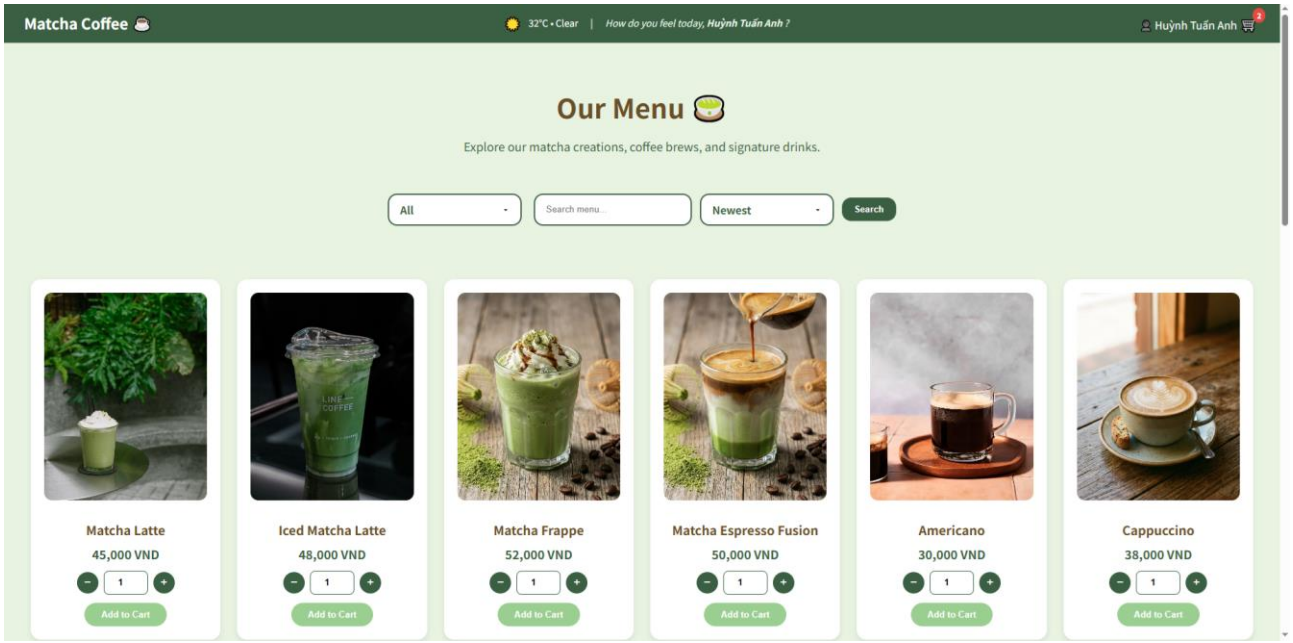


Explanation of the code flow of viewing the list of products (menu):

– Browser → GET /menu
+ The user clicks the Menu link or navigates to the menu page.
+ The browser sends a GET request to /menu.
+ Optional query parameters may include category (filter by category), q (search keyword), and sort (price sorting).
– MenuController receives request:
+ The doGet() method in MenuController is executed.
+ The controller retrieves request parameters category, q (search keyword), sort.
– Controller calls MenuDAO.getMenu(category, keyword, sort)
+ The controller delegates data retrieval to the DAO layer.
+ The controller does not handle SQL logic directly.
– MenuDAO builds dynamic SQL query
+ The base query selects only available products: SELECT * FROM menu WHERE status = 'available'.
+ Additional conditions are appended dynamically filter by category (if not “all”), search by product name (using LIKE), and sort by price or newest products.
– DAO prepares and executes SQL query
+ A PreparedStatement is created to prevent SQL injection.
+ Parameters are bound dynamically based on user input.
+ The query is executed against the MySQL database.
– DAO processes ResultSet:
+ For each row returned: A Menu object is created, and product attributes are populated: ID, name, description, price, image URL, category, status, created and updated timestamps.
+ Each Menu object is added to a List<Menu>.

– DAO returns List<Menu> to controller:
+ The list contains all products that match the filter, search, and sort conditions.
+ If no product matches, the list is empty.
– Controller sets request attributes:
+ The controller attaches data to the request menuList, selectedCategory, keyword, sort.
+ These attributes are used by the JSP for display and state preservation.
– Controller forwards request to menu.jsp:
+ Forwarding is used instead of redirecting to keep request attributes.
+ The JSP receives the product list and filter values.
– JavaScript initializes filter UI state:
+ JavaScript reads current URL parameters.
+ Category dropdown, search input, and sort selection are initialized correctly.
+ Hidden inputs ensure filter state is preserved between requests.
– User interacts with filters (category, search, sort):
+ Selecting a category automatically submits the form.
+ Changing sort order submits the search form.
+ Search keyword input allows name-based filtering.
+ All interactions trigger a new GET /menu request.
– JSP displays the menu list:
+ menu.jsp uses JSTL <c:forEach> to loop through menuList.
+ Each product card displays image, product name, price, quantity selector, and "Add to Cart" button.
– HTML response sent to browser:
+ The fully rendered menu page is returned to the browser.
+ Users can browse, filter, search, sort, and add products to cart

1.5.Add to cart from menu



Adding products to cart from menu and seeing the number of products in cart.

Explanation of the quantity increase/decrease flow on the menu page:

– User views the menu page: Each product card on menu.jsp contains a quantity input field, and two buttons: **"+"** (increase) and **"–"** (decrease).
– User clicks the "+" button:

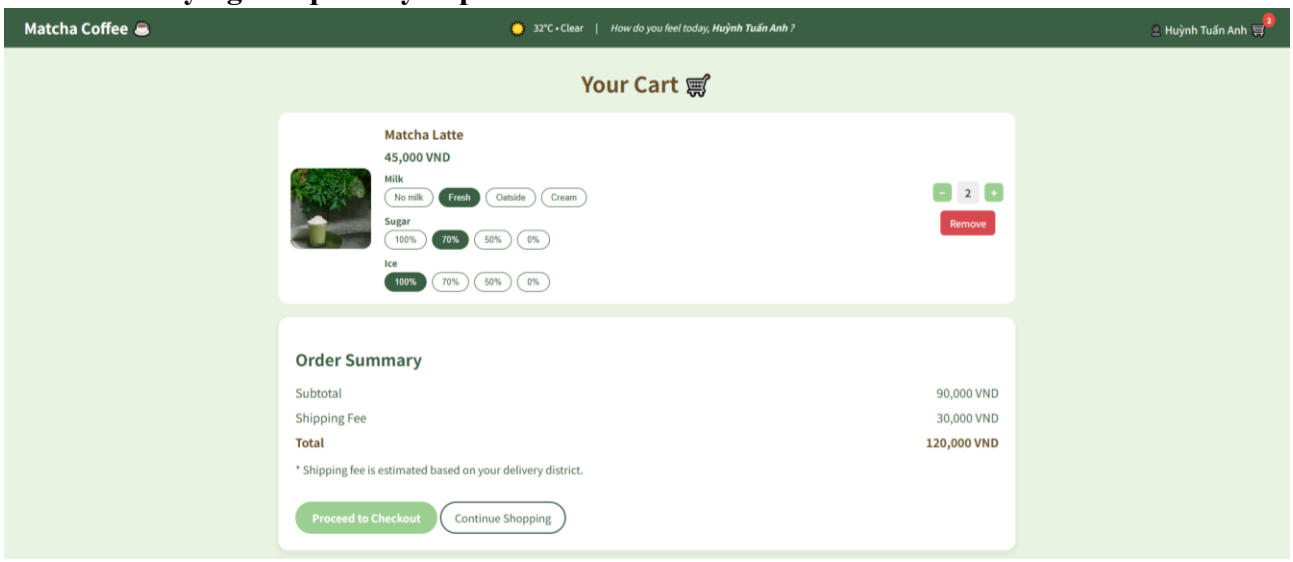
+ The increaseQty(btn) JavaScript function is triggered.
+ The clicked button is passed as a parameter.
– JavaScript locates the quantity input
+ The function finds the related quantity input field by accessing: btn.parentElement.querySelector("input").
+ This ensures the correct product quantity is modified.
– JavaScript reads the current quantity: The current value of the input field is retrieved and converted to an integer.
– Maximum quantity validation: The function checks if the quantity is less than 20. This prevents users from selecting an excessive quantity.
– Quantity is increased: If validation passes, the quantity is increased by 1. The updated value is displayed immediately on the UI.
– User clicks the "–" button: The decreaseQty(btn) JavaScript function is triggered.
– JavaScript locates the quantity input: Similar to the increase function, the related input field is located.
– Minimum quantity validation: The function checks if the quantity is greater than 1. This prevents the quantity from dropping below the minimum allowed value.
– Quantity is decreased: If validation passes, the quantity is reduced by 1. The UI is updated immediately.
– No server request is made:
+ Quantity changes happen entirely on the client side.
+ The server is only contacted when the user clicks "Add to Cart" button.
– Quantity value is submitted with the form
+ When the user adds the product to the cart, the selected quantity is sent to the server as a request parameter (qty).
+ This value is processed by CartController.

Explanation of the code flow of adding products to cart from menu:

– Browser → GET /menu
+ The user views the menu page.
+ Each product card contains an Add to Cart form with product ID, quantity, operation type
– User clicks "Add to Cart": The browser sends a GET request: /cart?op=add&pid={productId}&qty={quantity}
– CartController receives GET request:
+ The doGet() method in CartController is executed.
+ The controller retrieves the user session and reads userId.
– Authentication check:
+ If userId is null: The user is redirected to login.jsp.
+ Only logged-in users can add items to the cart.
– Controller detects add-to-cart operation:
+ The controller checks op = "add", pid is not null
+ The controller proceeds with add-to-cart logic.
– Controller reads product quantity and default options:
+ Quantity is read from request (default = 1).
+ Default options are set milk, sugar, ice for drinks, no options for bakery items.
– Controller calls MenuDAO.getMenuById(pid):
+ The controller retrieves product information from the database.
+ The DAO returns a Menu object containing price, category, product details.

– Controller calculates product price:
+ Base price is taken from the menu item.
+ Extra price is calculated based on milk selection:
- Oatside → +5000 VND
- Cream Milk → +7000 VND
- Fresh Milk / No milk → +0
+ Final price = base price + extra price
– Controller calls CartDAO.addToCart(): The controller passes userId, productId, quantity, price, drink options (milk, sugar, ice, toppings).
– CartDAO checks for existing cart item:
+ DAO executes a SELECT query to check same user, same product, same drink options.
+ This prevents duplicate rows for identical items.
– DAO inserts or updates cart item:
+ If item already exists quantity is increased.
+ If item does not exist, a new cart record is inserted into cart_items
– Controller updates cart item count:
+ The controller calls CartDAO.getTotalQuantity(userId).
+ Total quantity of all cart items is calculated.
+ cartCount is stored in the session.
– Controller redirects back to menu:
+ The user remains on the menu page.
+ The cart count badge is updated immediately.

1.6.Modifying the quantity of products in cart



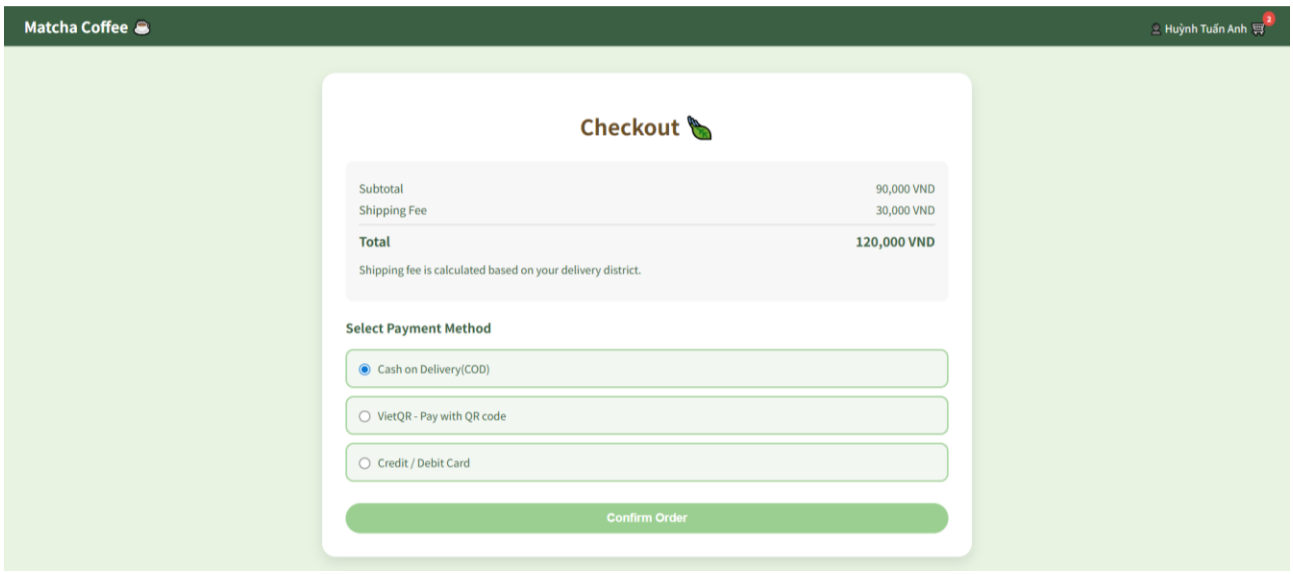
The cart view that shows all order items

Explanation of modifying the changes in milk, sugar, ice of products in cart:

– Browser loads the cart page
+ The cart page (cart.jsp) is loaded.
+ The DOMContentLoaded event is triggered once the HTML is fully rendered.
– JavaScript initializes toggle groups
+ The script selects all elements with class .toggle-group.
+ Each toggle group represents one option type milk, sugar, ice.
– JavaScript reads option metadata: For each toggle group:
+ The option name is read from data-name (e.g., milk, sugar, ice).

+ The nearest form element is located.
+ A hidden input matching the option name is selected.
– User clicks an option button:
+ Each option button (.toggle-btn) has a data-value.
+ When clicked, a JavaScript click event is triggered.
– JavaScript updates UI state:
+ The script removes the active class from all sibling buttons.
+ The clicked button is marked as active.
+ This provides immediate visual feedback to the user.
– JavaScript updates hidden input
+ The hidden input value is updated to match the selected option.
+ This ensures the selected value is included in the form submission.
– Automatic form submission:
+ After a short delay (150 ms): The form is automatically submitted.
+ This creates a smooth user experience without requiring a submit button.
– Browser → POST /cart: The form submission sends a POST request to /cart with parameters op = updateOptions, cid (cart item ID), milk or sugar or ice.
– CartController receives POST request:
+ The doPost() method is executed.
+ The controller retrieves user session, user ID, operation type (updateOptions)
– Controller validates update operation:
+ The controller checks if op = updateOptions.
+ Only option update logic is executed.
– Controller calls CartDAO.updateOptions():
+ The controller passes updated values to the DAO layer.
+ This keeps business logic separate from UI logic.
– DAO validates product type:
+ If the product is a bakery item: The update is ignored.
+ Drink options are processed normally.
– DAO recalculates pricing:
+ Extra price is recalculated based on milk selection.
+ Final price is updated automatically.
– DAO updates database:
+ The DAO executes an SQL UPDATE query.
+ Changes persisted immediately.
– Controller redirects back to /cart:
+ Redirection prevents duplicate submissions.
+ The updated cart page is reloaded.
– Cart page reloads with updated values
+ Updated options, prices, and totals are displayed.
+ Cart quantity badge remains synchronized.

1.7.Choosing payment methods at checkout page



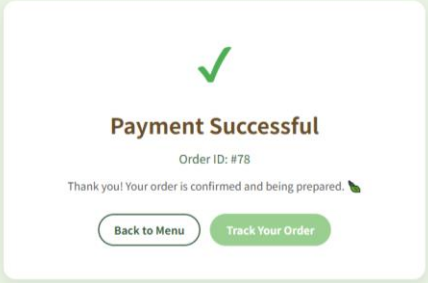
The checkout page and payment methods

Explanation of the code flow of showing the checkout page:

- Browser → GET /checkout
+ The user clicks the **Checkout** button from the cart page.
+ A GET request is sent to /checkout.
- OrderController receives GET request
+ The doGet() method in OrderController is executed.
+ The controller retrieves the current HTTP session.
+ The userId is obtained from the session.
- Controller loads cart items
+ The controller calls CartDAO.getCartByUser(userId).
+ All cart items belonging to the user are loaded from the database.
- Empty cart validation:
+ If the cart is empty: The user is redirected back to /cart.
+ This prevents accessing the checkout page without items.
- Controller calculates subtotal and total cups
+ The controller iterates through all cart items:
- subtotal += finalPrice x quantity
- cups += quantity
+ This represents the total product cost and number of drinks.
- Controller calculates shipping fee
+ The controller retrieves the logged-in user from the session.
+ The full address is extracted from currentUser.
+ AddressUtil.extractDistrict() extracts the district.
+ ShippingService.calculateShipping(city, district) is called.
+ A shipping fee is returned based on location.
- Controller calculates total amount: total = subtotal + shipping. This represents the final payable amount.
- Controller sets request attributes: The following data is attached to the request cart (list of cart items), subtotal, shipping, total, cups
- Controller forwards to checkout.jsp
+ The request is forwarded (not redirected).
+ All calculated values are preserved for rendering.

– JSP renders order summary: checkout.jsp displays subtotal, shipping fee, total amount.
– JSP displays cart badge
+ The navbar displays the cart icon.
+ The cart badge shows cartCount from the session.
– JSP displays payment method options:
+ The user is shown available payment methods such as cash on delivery (COD), VietQR, Credit/Debit Card.
+ COD is selected by default.

1.7.1. Cash on delivery (COD)



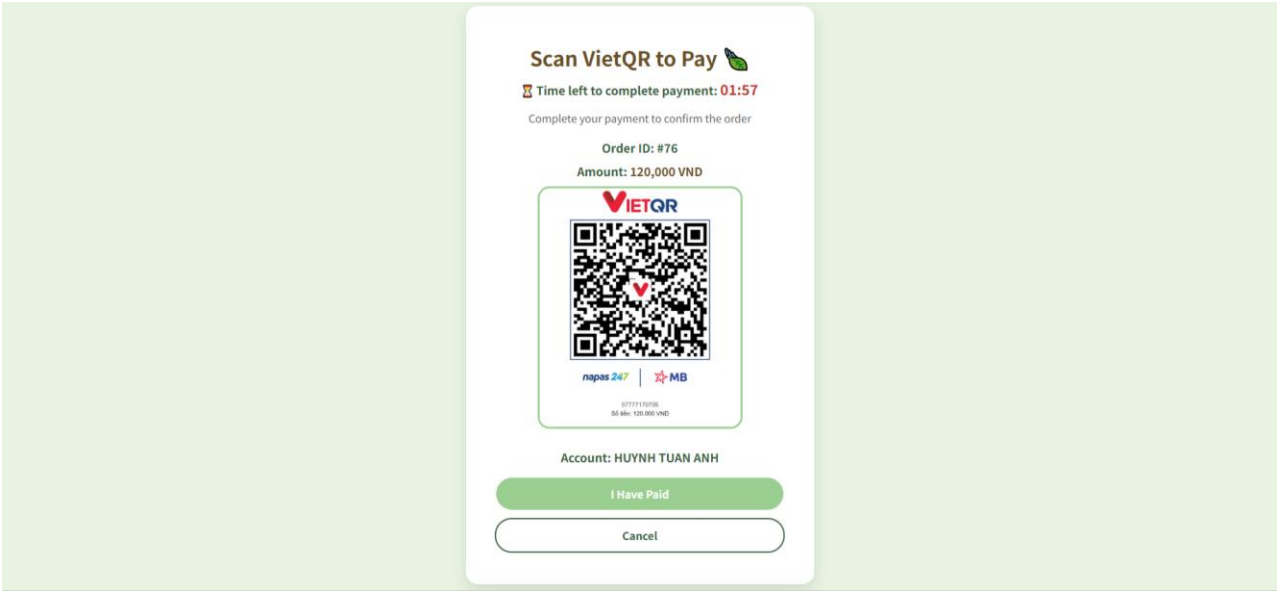
The payment result page after choosing COD as a payment method

Explanation of the flow of the cash on delivery as payment method:

– User views the checkout page
+ The user navigates to /checkout.
+ Order summary (subtotal, shipping fee, total) is displayed.
+ The user selects Cash on Delivery (COD) as the payment method.
– User confirms the order
+ The user clicks the Confirm Order button.
+ A POST request is sent to /checkout with: paymentMethod = COD
– OrderController receives POST /checkout
+ The doPost() method is executed.
+ The controller retrieves userId from session, paymentMethod from request
– Controller validates payment method: If paymentMethod is missing, the user is redirected back to the checkout page with an error.
– Controller loads cart data:
+ CartDAO.getCartByUser(userId) is called.
+ If the cart is empty, the user is redirected back to the cart page.
– Controller calculates order values:
+ Subtotal is calculated: finalPrice × quantity for each cart item.
+ Total number of cups is counted.
+ Shipping fee is calculated using ShippingService: Based on city and extracted district from user address.
+ Total amount = subtotal + shipping fee.
– Controller creates Order object: A new Order object is created with userId, subtotal, shipping fee, total amount, total cups, payment method is COD.
– Order status and payment status are set
+ Since the payment method is COD: order.status = "confirmed", order.paymentStatus = "unpaid".
+ This indicates the order is accepted but payment will be collected upon delivery.
– Order is saved to the database
+ OrderDAO.saveOrder(order) is called.

+ The order is inserted into the orders table.
+ The generated orderId is returned.
– Cart items are moved to order items:
+ OrderDAO.moveCartToOrderItems(orderId, cart) is executed.
+ All cart items are copied into the order_items table.
+ This step finalizes the order contents.
– Shipping information snapshot is created:
+ A Shipping object is created with orderId, receiver name, phone number, address, city, district, shipping fee, shipping method and status
+ ShippingDAO.createShipping() saves this snapshot.
+ This ensures shipping details are preserved even if the user updates their profile later.
– Controller clears the cart:
+ CartDAO.clearCart(userId) is called.
+ All cart items for the user are removed.
+ cartCount in session is set to 0.
– Controller redirects to payment result page: The user is redirected to: /payment-result?status=success&orderId={orderId}.
– PaymentResultController receives GET /payment-result: The controller retrieves status, orderId, userId from session
– Final cart cleanup and confirmation: If status = success and user is authenticated:
+ Cart is ensured to be cleared.
+ Cart badge remains at 0.
– Controller forwards to payment-result.jsp: The result page display orderId, payment status, confirmation message for COD order.

1.7.2. Pay with QR code (VietQR)



The VietQR payment method with QR code and countdown time

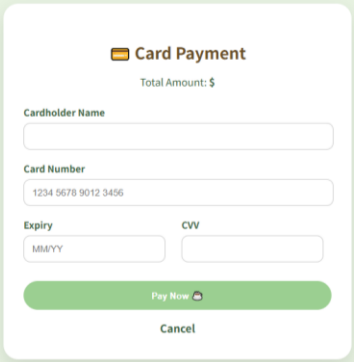
Explanation of the code flow of scanned QR code with VietQR as payment method:

– User selects VietQR as payment method: On the checkout page, the user selects VietQR and clicks Confirm Order.
– Browser → POST /checkout: The checkout form is submitted to OrderController.doPost().
– OrderController processes VietQR payment
+ The controller loads cart items, calculates subtotal, shipping fee, and total amount, creates a new Order object, set status is pending and paymentStatus is pending.

– Order is saved to database
+ orderDAO.saveOrder(order) is called.
+ The database generates a new orderId.
– Cart items are moved to order items
+ orderDAO.moveCartToOrderItems(orderId, cart) transfers cart data.
+ Cart data is preserved in order history.
– Shipping snapshot is created
+ A Shipping object is created using the user's current address.
+ Shipping information is saved via ShippingDAO.createShipping().
– Controller redirects to VietQR page: The browser is redirected to: /vietqr?orderId={orderId}
– Browser → GET /vietqr: The request is handled by VietQRController.doGet().
– VietQRController loads order information:
+ The controller retrieves the order using orderId.
+ The total payment amount is extracted from the order.
– QR code URL is generated: A VietQR image URL is constructed using bank code, account number, order amount, order ID as transfer description.
– Controller sets request attributes: Attributes passed to vietqr.jsp: qrUrl, orderId, amount, createAt, accountName.
– Controller forwards to vietqr.jsp: The VietQR payment page is rendered.
– VietQR page is displayed: The page show Order ID, total amount, VietQR image, bank account name, and countdown timer.
– Order metadata is passed to JavaScript:
+ ORDER_CREATED_AT and ORDER_ID are injected into JavaScript variables.
+ These values control payment expiration.
– JavaScript initializes countdown
+ countdown.js runs after page load.
+ Order creation time is converted into a JavaScript Date.
– Expiry time is calculated
+ Payment window = 2 minutes
+ Expiry time = createdAt + 2 minutes
– Countdown updates every second
+ Remaining time is displayed in mm:ss format.
+ UI updates in real time without server calls.
– Countdown reaches zero: If time expires:
+ Countdown stops
+ Browser is automatically redirected to: payment-result?status=failed&orderId={orderId}
– User scans QR code using banking app: User completes payment manually through their bank app.
– User clicks "I Have Paid":
Browser navigates to: payment-result?status=success&orderId={orderId}
– User clicks "Cancel":
Browser navigates to: payment-result?status=failed&orderId={orderId}
– PaymentResultController receives request: The controller reads status, orderId, user session
– Successful payment handling: If status = success, cart is cleared, cartCount is set to 0, and order remains in pending state (admin confirmation required).
– Failed or cancelled payment: If status = failed, cart is not cleared, order remains pending or cancelled based on admin handling.

- Result page is displayed: payment-result.jsp shows the final payment status to the user.

1.7.3. Pay with credit/debit card



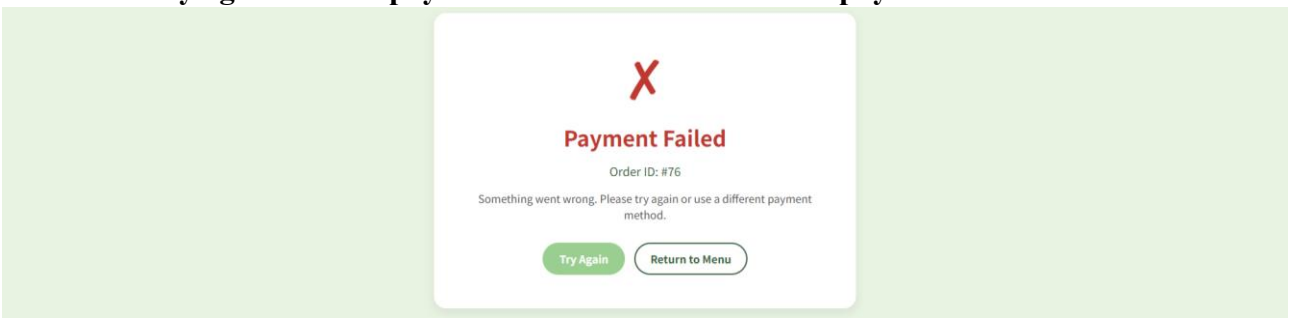
The card payment view with the input fields of card information

Explanation of the code flow of the card as payment method:

- User selects Card as payment method
+ On checkout.jsp, the user selects: Credit / Debit Card
+ The checkout form is submitted with: paymentMethod = CARD
- Browser → POST /checkout
+ The browser sends a POST request to /checkout.
+ The request contains the selected payment method.
- OrderController receives POST request:
+ The doPost() method in OrderController is executed.
+ The controller retrieves userId from session, paymentMethod from request
- Controller validates payment method:
+ If paymentMethod is missing or empty: The user is redirected back to checkout.jsp with an error.
+ If valid, processing continues.
- Controller loads cart items:
+ The controller calls: cartDAO.getCartByUser(userId)
+ If the cart is empty: The user is redirected back to the cart page.
+ This prevents creating orders without items.
- Controller calculates order values:
+ For each cart item: finalPrice × quantity is accumulated.
+ The following values are calculated subtotal, total cups.
+ Shipping fee is calculated using user address, ShippingService.
+ Final total: total = subtotal + shipping
- Controller creates Order object:
+ A new Order object is created.
+ The following data is set user ID, subtotal, shipping fee, total amount, total cups, payment method = CARD.
- Controller sets initial order and payment status:
+ Because payment is online and not yet completed: order.status = "pending", order.paymentStatus = "pending".
+ This indicates the order is waiting for card payment confirmation.
- Order is saved to database:
+ The controller calls: orderDAO.saveOrder(order)
+ The database returns a generated orderId.
- Cart items are moved to order_items:

+ The controller calls: orderDAO.moveCartToOrderItems(orderId, cart)
+ This step preserves cart data at checkout time and prevents art changes after order creation.
– Shipping snapshot is created:
+ A Shipping object is created with receiver name, phone, address, city and district, shipping fee, shipping status = pending.
+ This snapshot ensures shipping details remain unchanged even if user updates profile later.
– Controller redirects to card payment page:
+ Based on payment method: CARD → card-payment.jsp?orderId={orderId}
+ The user is redirected to the card payment interface.
– User completes card payment (external or simulated): On the card payment page:
+ User enters card details
+ Payment is processed (real or simulated)
– Payment confirmation updates order:
+ After successful card payment: Order payment status is updated to PAID, and order status is updated to CONFIRMED.
+ This update is typically handled by a confirmation controller or admin process.
– Cart is cleared after successful payment: Once payment is confirmed:
+ Cart items are removed from the database.
+ cartCount in session is reset to 0.
– User is redirected to payment result page
+ The user is redirected to: /payment-result?status=success&orderId={orderId}
+ The result page displays payment status, order ID, and confirmation message.

1.7.4. Try again to with payment failed and choose other payment methods



The payment failed display and try again button

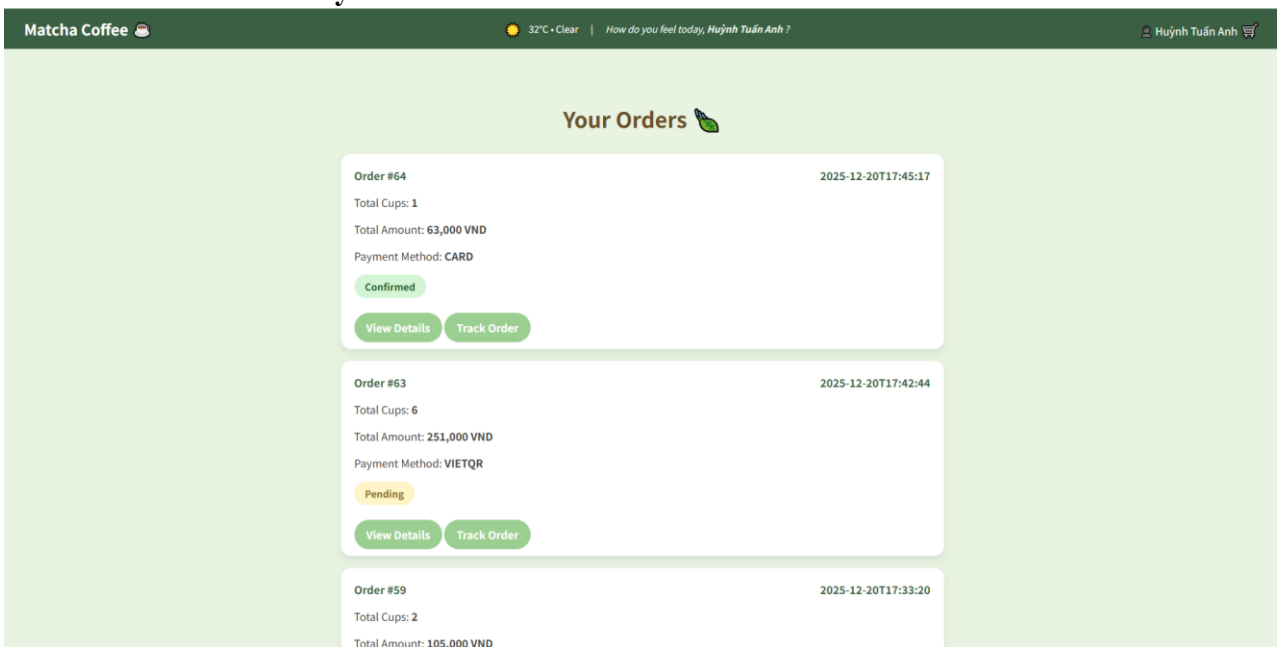
Explanation of the code flow of trying again checkout:

– User completes checkout but payment fails or is cancelled
+ This can happen when VietQR countdown expires, user clicks cancel on VietQR page.
+ The browser is redirected to /payment-result?status=failed&orderId={orderId}
– PaymentResultController receives GET request:
+ The doGet() method in PaymentResultController is executed.
+ The controller retrieves status (failed), orderId, userId from session.
– Controller does NOT clear the cart
+ Because status != success: Cart items are not deleted, cartCount remains unchanged
+ This allows the user to retry checkout.
– Controller forwards to payment-result.jsp: The page displays payment failed message, order ID, available actions such as (Try again, back to cart).
– User clicks “Try Again”:
+ The user chooses to retry the checkout process.
+ The browser navigates back to: /checkout

- Browser → GET /checkout: A new GET request is sent to the checkout endpoint.
- OrderController receives GET request
+ The doGet() method in OrderController is executed.
+ The controller (OrderController.java file) retrieves userId from session, and the cart items using CartDAO.getCartByUser(userId).
- Controller validates cart state:
+ If the cart is empty: User is redirected to /cart
+ If the cart still has items: Checkout process continues normally
- Controller recalculates checkout values:
+ Subtotal is recalculated from cart items
+ Shipping fee is recalculated based on user address, district, city
+ Total amount is recalculated.
- Controller sets checkout attributes: The following attributes are attached to the request cart, subtotal, shipping, total, cups.
- Controller forwards to checkout.jsp:
+ Forwarding preserves calculated values.
+ The checkout page is displayed again.
- User selects payment method again:
+ The user may choose a different payment method (COD/VietQR/Card) or retry the same payment method.
+ The checkout form is submitted again.
- Browser → POST /checkout
+ A new order attempt is initiated.
+ The normal checkout submission flow resumes.

1.8.View orders

1.8.1. View all history orders



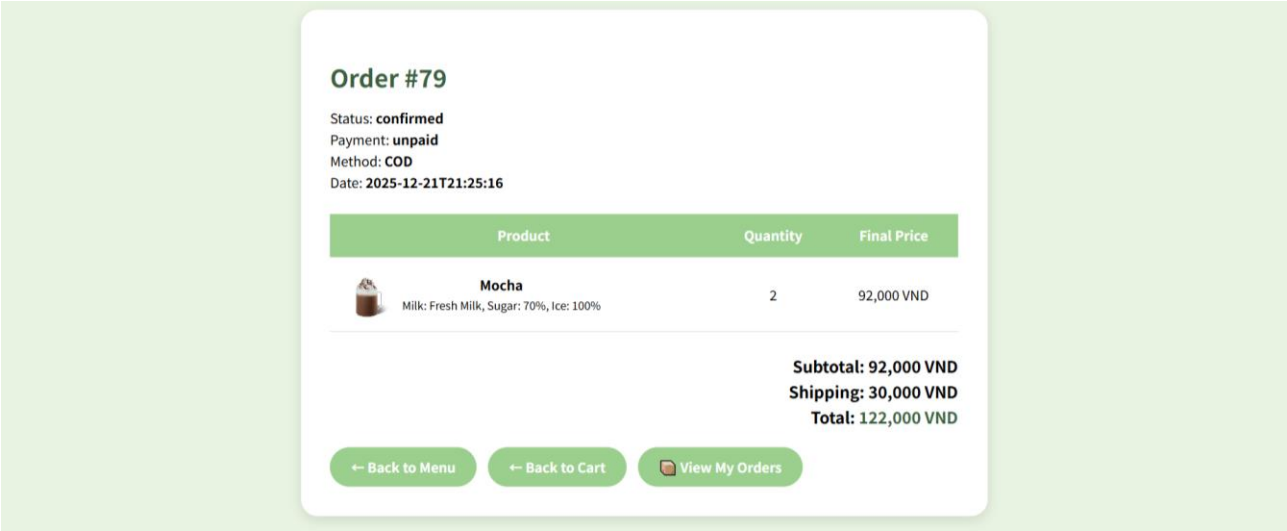
Viewing all history orders

Explanation of the code flow of viewing all history orders:

- Browser → GET /my-orders
+ The user clicks **Orders** in the navigation bar.
+ The browser sends a GET request to /my-orders.

– MyOrdersController receives request
+ The doGet() method in MyOrdersController is executed.
+ The controller retrieves the current HTTP session.
– Controller retrieves logged-in user ID
+ The controller reads userId from the session.
+ This ensures that only the logged-in user's order history is accessed.
– Controller calls OrderDAO.getOrdersByUser(userId)
+ The controller delegates data access to the DAO layer.
+ The controller does not directly execute SQL queries.
– OrderDAO queries the database:
+ The DAO executes the SQL query: SELECT * FROM orders WHERE user_id = ? ORDER BY created_at DESC.
+ Orders are sorted so the newest orders appear first.
– DAO builds list of Order objects: For each row returned from the database:
+ An Order object is created.
+ Order details are populated order ID, subtotal, shipping fee, total amount, total cups, order status, payment method, payment status, created date.
+ Each Order object is added to a List<Order>.
– DAO returns List<Order> to controller:
+ The list contains all historical orders of the current user.
+ If the user has no orders, the list is empty.
– Controller sets request attribute: The controller attaches the order list to the request: request.setAttribute("orders", orders);.
– Controller forwards to my-orders.jsp
+ Forwarding (not redirect) is used to preserve request data.
+ The JSP page receives the order list.
– JSP checks for empty order list: If the list is empty: A message such as *"You have no orders yet"* is displayed.
– JSP iterates through orders: <c:forEach> loops through the orders list. Each order card displays order ID, order creation date, total cups, total amount, payment method.
– JSP displays order status:
+ <c:choose> is used to display status visually:
- Confirmed → confirmed status
- Pending → pending status
- Cancelled → cancelled status
+ Each status is styled differently for clarity.
– JSP provides order actions: Each order includes buttons:
+ View Details → view items in the order
+ Track Order → check shipping status
– HTML response sent to browser:
+ The fully rendered order history page is returned.
+ The user can review all past orders and their statuses.

1.8.2. View order in details:



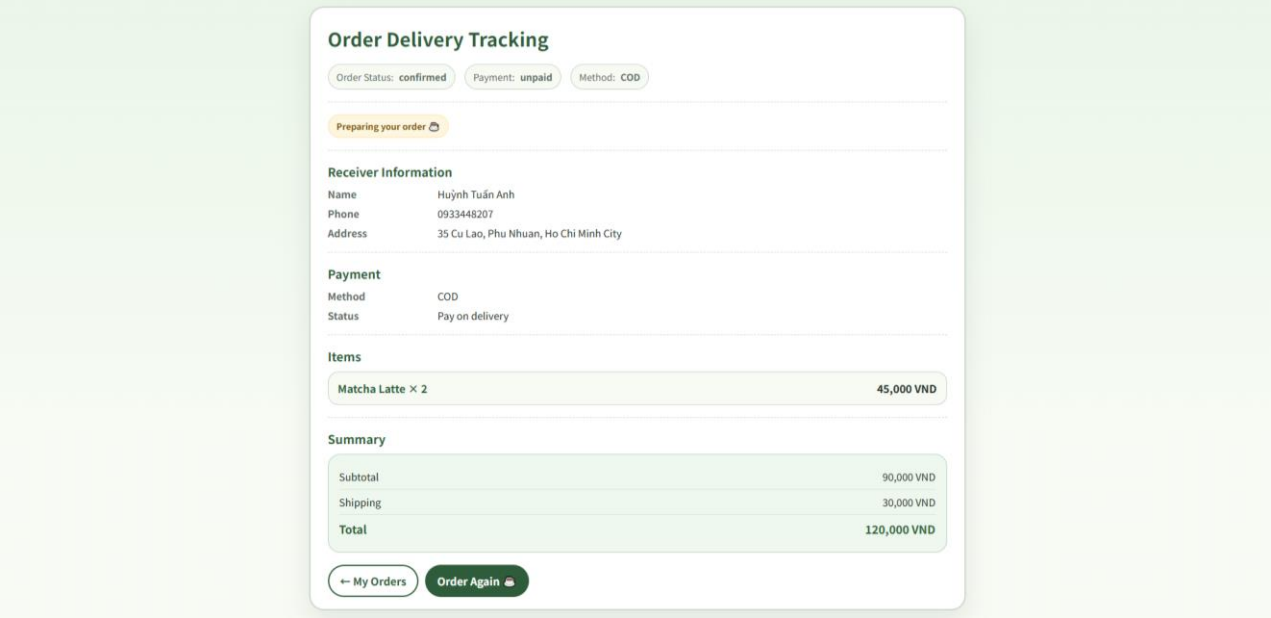
View order in details

Explanation of code flow of viewing order in details:

- Browser → GET /order-details?id={orderId}
+ The user clicks “View Details” from the My Orders page.
+ The browser sends a GET request with the selected orderId.
- OrderDetailsController receives request
+ The doGet() method of OrderDetailsController is executed.
+ The controller retrieves the id parameter from the request.
- Controller validates request parameter
+ If id is missing: The user is redirected to menu.jsp.
+ This prevents invalid or direct access without an order ID.
- Controller converts order ID
+ The id parameter is converted from String to int.
+ This value represents the unique order identifier in the database.
- Controller retrieves order information:
+ The controller calls OrderDAO.getOrderById(orderId).
+ The DAO queries the orders table to retrieve order status, payment status, payment method, subtotal, shipping fee, total amount, order creation date.
- Controller retrieves order items:
+ The controller calls OrderDAO.getOrderItems(orderId).
+ The DAO joins order_items with menu to fetch product name, image, quantity, price, drink options (milk, sugar, ice).
- DAO returns data to controller
+ A single Order object is returned.
+ A List<OrderItem> containing all products in the order is returned.
- Controller sets request attributes: The controller attaches data to the request:
+ order → order summary information
+ items → list of order items
- Controller forwards request to order-details.jsp
+ Forwarding is used to preserve request attributes.
+ The JSP receives all order data for rendering.
- JSP displays order summary
+ Order ID, status, payment status, and payment method are shown.
+ Order creation date is displayed.

– JSP displays order items
+ The JSP uses <c:forEach> to loop through items.
+ Each row displays product image and name, quantity, final price x quantity, drink options (milk, sugar, ice) if applicable.
– JSP displays order totals: subtotal, shipping fee, total amount.
– Navigation actions available: Buttons allow the user to return to menu, return to cart, view all orders.
– HTML response sent to browser: The fully rendered order details page is displayed to the user.

1.9.Tracking order for shipping



The view of tracking order

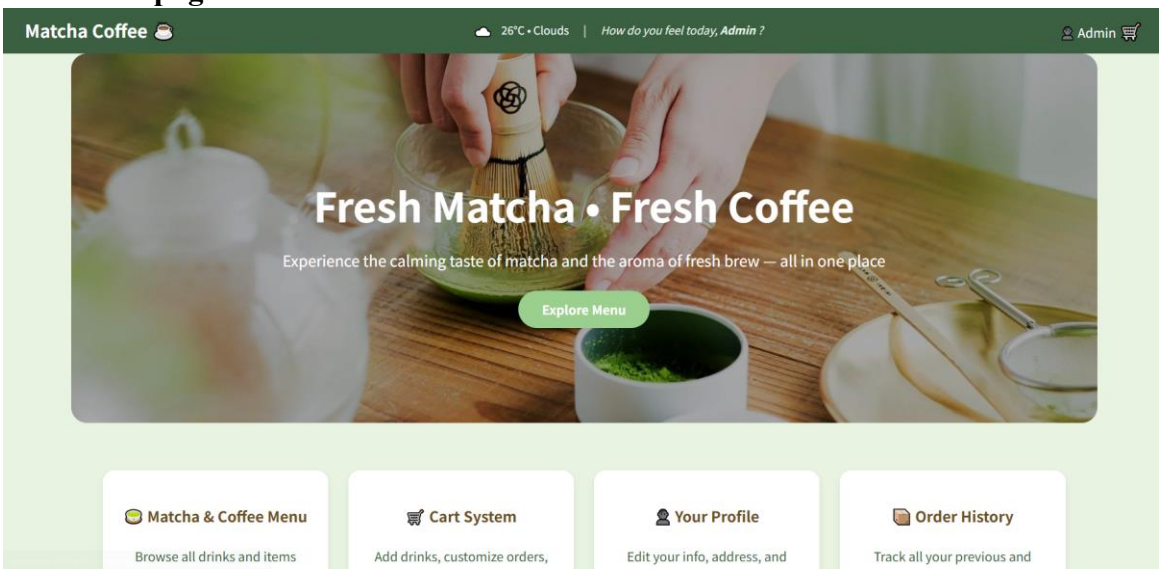
Explanation of the code flow of the viewing the order delivery tracking page:

– Browser → GET /shipping?orderId={id}: The user clicks "Track Order" from: My Orders page, or Order details page. The browser sends a GET request to /shipping with the orderId parameter.
– ShippingController receives request
+ The doGet() method in ShippingController is executed.
+ The controller reads the orderId from the request parameters.
– Controller retrieves order information
+ The controller calls: orderDAO.getOrderById(orderId)
+ This retrieves general order data order status, payment method, payment status, total amount, created date.
– Controller retrieves shipping information
+ The controller calls: shippingDAO.getShippingByOrderId(orderId)
+ This retrieves shipping snapshot data saved at checkout receiver name, phone number, address, city, district, shipping fee, shipping method, shipping status.
– Controller retrieves order items
+ The controller calls: orderDAO.getOrderItems(orderId)
+ This returns a list of items belonging to the order product name, quantity, and final price per item
– Controller sets request attributes: The following attributes are attached to the request:
+ order → order metadata
+ shipping → delivery information

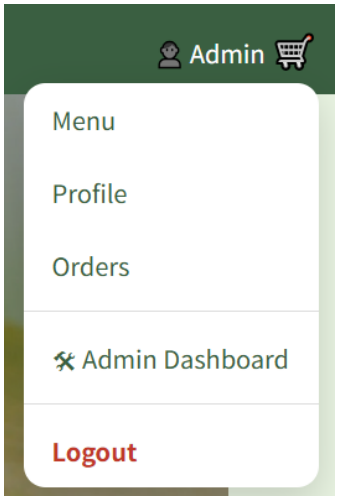
+ orderItems → list of ordered products
– Controller forwards to shipping.jsp
+ Forwarding is used to preserve request data.
+ The request is forwarded to the delivery tracking page.
– JSP displays order and payment status: shipping.jsp display order status, payment status, payment method
– JSP displays dynamic delivery status: JSTL <c:choose> determines the delivery message
– JSP displays receiver information: Receiver details are shown name, phone number, address.
– JSP displays ordered items: <c:forEach> loops through orderItems. Each item displays product name, quantity, price
– JSP displays order summary: The summary section shows subtotal, shipping fee, total amount.
– User navigation actions. The user can go back to My Orders or click Order Again to return to the menu.
– HTML response sent to browser: The fully rendered delivery tracking page is returned. The user can monitor the real-time delivery state.

2. Admin's account permission

2.1.View homepage



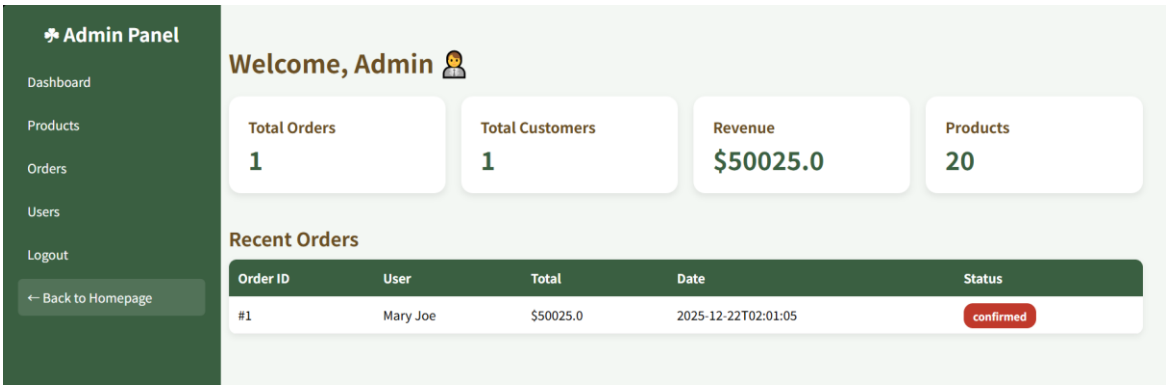
2.2.View dashboard



Explanation of the code flow when admin clicks on 'admin dashboard':

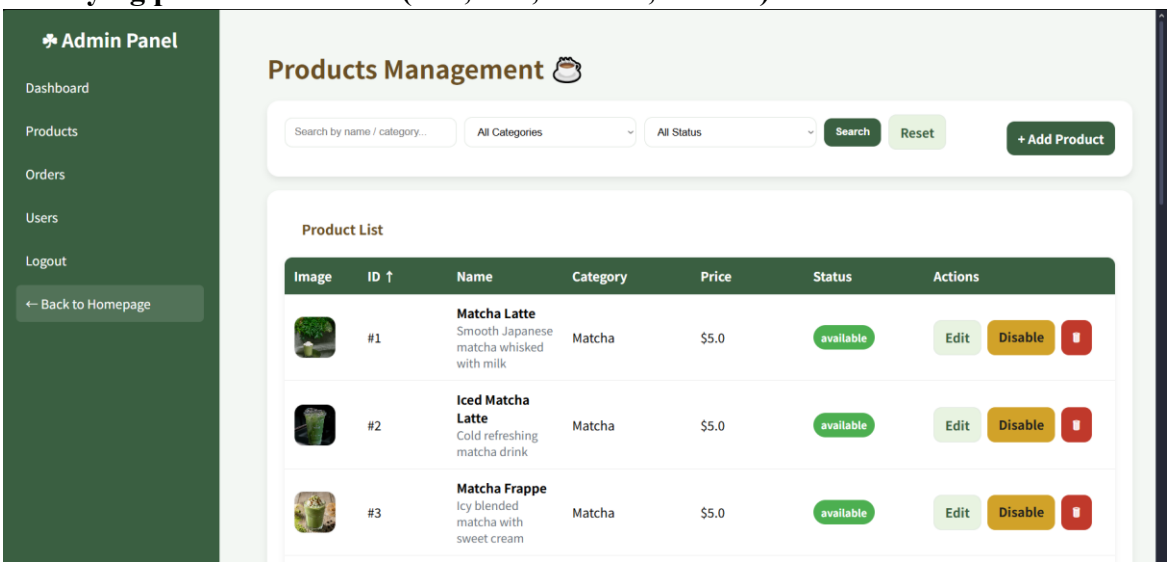
- Browser → GET /admin/dashboard
- The admin is currently on the homepage of the website
- The options will drop down when the cursor is moved to the admin icon
- The browser sends a GET request to the dashboard route mapped in web.xml
- Filters run before the controller
- AdminFilter / AdminAuthFilter intercepts the request
- The filter checks:
  - User is logged in (session exists).
  - User role is admin (not customer).
- If not logged in / not admin:

        - Redirect to login.jsp or show access denied.
    - If valid admin:
        - The request continues to the controller.
- AdminDashboardController receives request
    - AdminDashboardController.doGet() is executed.
- Controller retrieves dashboard summary data
    - The controller calls DAO methods to load the numbers shown on the dashboard.
    - These DAO methods query tables such as orders, order_items, users, and return totals / lists.
- DAO returns data to controller
    - Controller receives:
        - Summary metrics (counts, totals)
        - Optional lists
- Controller sets request attributes
    - The controller attaches dashboard data to the request.
- Controller forwards to admin-dashboard.jsp
    - Uses RequestDispatcher.forward() to keep request attributes available.
    - Example target: /admin-dashboard.jsp.
- JSP renders the dashboard UI
    - admin-dashboard.jsp reads the request attributes.
    - It displays:
        - KPI cards (totals)
        - Tables/lists (recent orders/users)
        - Any status summaries (if present)
    - If the JSP uses JSTL:
        - <c:if> / <c:choose> for conditional blocks
        - <c:forEach> for lists like recent orders
- HTML response sent to browser
    - The fully rendered Admin Dashboard page is returned.
    - The admin now sees the dashboard data and can navigate to admin-orders.jsp, admin-users.jsp, etc.



Viewing the admin dashboard

2.3.Modifying products to menu (add, edit, remove, disable)



Viewing the products

Explanation of the code flow when admin clicks on 'products':

When an administrator navigates from the Admin Dashboard to the Products Management page, the system handles the request through authentication filters, controller processing, data retrieval, and view rendering.

- **Request initiation**

The administrator is currently logged in and viewing the *Admin Dashboard* page (admin-dashboard.jsp).

The administrator clicks the **"Products"** button or menu item in the admin navigation interface.

The browser sends a **GET request** to the /products endpoint, which is mapped to AdminProductsController.

- **Request filtering and authorization**

Before reaching the controller, the request is intercepted by the administrative authentication filter.

The filter verifies that:

o A valid user session exists.

o The authenticated user has the **admin** role.

If either condition fails, the request is rejected and the user is redirected to the login page or shown an access-denied response.

If both conditions are satisfied, the request is forwarded to the controller.

- **Controller processing**

The AdminProductsController receives the request and executes its doGet() method.

The controller first checks whether any administrative operation parameters (such as op or id) are present.

Since the request originates from the dashboard navigation, no operation parameters are detected, and the controller proceeds with loading the product list.

- **Controller reads request parameters**

The controller retrieves optional parameters related to product listing, including:

o Search keyword (q)

o Product category (category)

o Product availability status (status)

o Sorting options (sortBy, sortDir)
If these parameters are not provided, default values are applied.
• **Controller handles pagination logic**
The controller reads the page index from the request, if present.
A fixed page size is applied to limit the number of products displayed per page.
The database query offset is calculated based on the current page.
• **Controller requests data from DAO**
The controller calls DAO methods to retrieve product-related data, including:
o The total number of products matching the current filters.
o The paginated list of products to be displayed.
The DAO constructs SQL queries incorporating filtering, sorting, and pagination logic and executes them using prepared statements.
• **DAO returns data to controller**
The DAO returns the following data to the controller:
o A list of product objects.
o The total product count required for pagination calculations.
• **Controller prepares data for the view**
The controller attaches the retrieved data and relevant metadata (such as current page and sorting options) to the request as attributes.
• **Controller forwards request to admin-products.jsp**
The controller forwards the request to admin-products.jsp using RequestDispatcher.forward().
This approach ensures that all request attributes remain available for rendering the view.
• **Response rendering**
The admin-products.jsp page reads the request attributes and renders the Products Management interface.
The administrator is presented with the product list, management actions, and navigation controls.
• **Final response**
The fully rendered Products Management page is returned to the browser.
The administrator can now view, disable, delete, or further manage products from the admin interface.

Explanation of the code flow when admin clicks on 'remove' or 'disable' a product:
1. Code Flow for Disabling (Enabling) a Product
When an administrator disables or enables a product, the system performs a controlled update of the product's availability status in the database.

- Request initiation
  The administrator is on the *Products Management* page and selects the Disable (or Enable) action for a specific product.
  The browser sends a GET request to the /products endpoint, including parameters that indicate the requested operation (op=toggle) and the target product identifier.
- Request filtering and authorization
  Before the request reaches the controller, it is intercepted by the administrative authentication filter.
  The filter verifies that:
  - A valid user session exists.

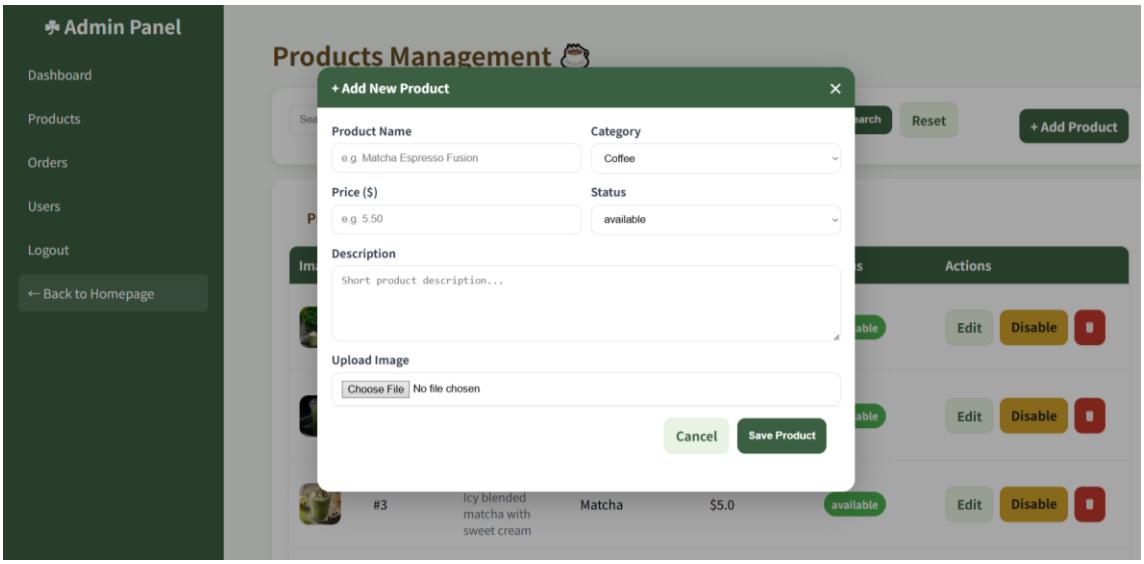
    - The logged-in user has the admin role.
      If either condition fails, the request is rejected and the user is redirected to the login page or shown an access-denied response.
      If the checks succeed, the request is forwarded to the controller.
- Controller processing
  The AdminProductsController receives the request and executes its doGet() method.
  The controller inspects the request parameters and determines that the operation type is a status toggle.
  It extracts the product identifier from the request and delegates the business logic to the data access layer.
- Data access and database update
  The controller invokes the DAO method responsible for toggling product availability.
  The DAO constructs and executes a prepared SQL update statement that changes the product's status between *available* and *unavailable*.
  Prepared statements are used to ensure data integrity and prevent SQL injection.
- Post-processing and redirection
  After the database update is completed, control returns to the controller.
  The controller issues a redirect back to the Products Management page, preserving any existing filters, sorting options, or pagination state.
  This ensures that the user interface reflects the updated product status.
- Response rendering
  A new request is triggered to reload the product list.
  The updated availability status is retrieved from the database and displayed to the administrator.

2. Code Flow for Deleting a Product

Deleting a product results in the permanent removal of the product record from the database.

- Request initiation
  From the *Products Management* page, the administrator selects the Delete action for a product.
  The browser sends a GET request to the /products endpoint with parameters specifying the delete operation (op=remove) and the product identifier.
- Request filtering and authorization
  The request is intercepted by the administrative filter.
  The filter confirms that the user is authenticated and has administrator privileges.
  Unauthorized requests are blocked, while valid requests are forwarded to the controller.
- Controller processing
  The AdminProductsController processes the request within its doGet() method.
  Upon detecting a delete operation, the controller extracts the product identifier and delegates the deletion task to the DAO layer.
- Data access and database deletion
  The DAO executes a prepared SQL DELETE statement targeting the product record associated with the provided identifier.
  Once executed successfully, the product is permanently removed from the database.
- Post-processing and redirection
  The DAO returns a success or failure status to the controller.
  The controller then redirects the user back to the Products Management page.
- Response rendering
  The product list is reloaded from the database.

Since the deleted product no longer exists, it is not included in the rendered results.
The administrator is presented with the updated product list reflecting the deletion.



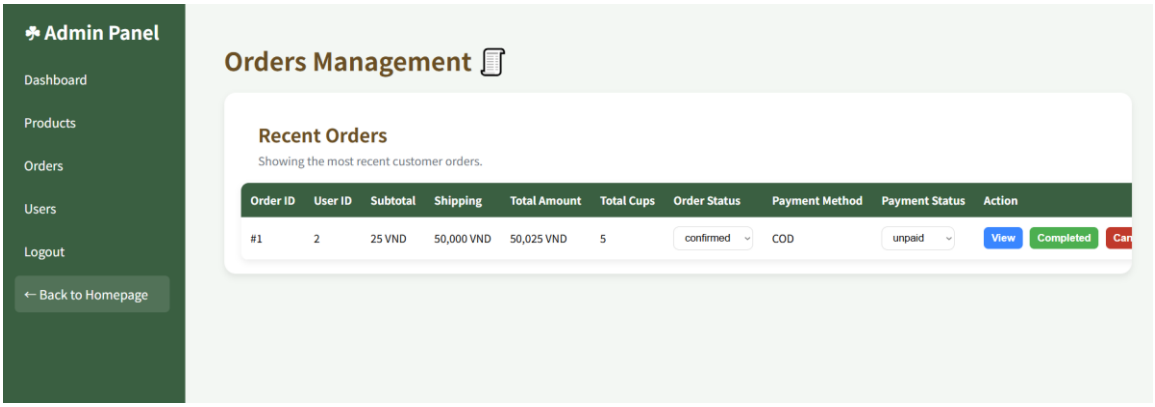
Explanation of the code flow when admin clicks on 'edit' and 'save changes' a product:
When an administrator edits a product, the system loads the existing product data, allows modifications, validates the input, and updates the database accordingly.

• Request initiation (Edit action)
The administrator is currently viewing the *Products Management* page.
The administrator clicks the Edit button for a specific product.
The browser sends a GET request to the /products (or /admin/products/edit) endpoint with the product identifier as a request parameter.
• Request filtering and authorization
Before reaching the controller, the request is intercepted by the administrative authentication filter.
The filter verifies that:
o A valid user session exists.
o The authenticated user has the admin role.
If either condition fails, the request is rejected and the user is redirected to the login page or shown an access-denied response.
If the checks succeed, the request is forwarded to the controller.
• Controller processing (Edit request)
The AdminProductsController (or dedicated EditProductController) receives the request and executes its doGet() method.
The controller extracts the product identifier from the request.
The controller delegates data retrieval to the DAO layer.
• Data access and product retrieval
The DAO executes a prepared SQL SELECT statement to retrieve the product record associated with the given identifier.
The database row is mapped to a Menu (or Product) model object.
The product object is returned to the controller.
• Controller forwards to edit-product.jsp
The controller attaches the product object to the request as an attribute.
The request is forwarded to edit-product.jsp using RequestDispatcher.forward().
This allows the existing product data to be prefilled in the edit form.

• Response rendering (Edit form)
The edit-product.jsp page renders the product edit form.
All editable fields (name, category, price, description, status, etc.) are populated with current product values.
The administrator modifies the product information and clicks Save Product.

• Request initiation (Save Product action)
After editing, the administrator submits the form.
The browser sends a POST request to the /products (or /admin/products/update) endpoint containing the updated product data.
• Request filtering and authorization
The POST request is intercepted by the administrative authentication filter.
The filter verifies that the user is authenticated and has admin privileges.
If authorization fails, the request is blocked.
If authorization succeeds, the request proceeds to the controller.
• Controller processing (Save request)
The controller executes its doPost() method.
All submitted form parameters are retrieved from the request.
The controller performs basic validation on required fields (e.g., product name, price, category).
If validation fails, an error message is attached to the request and the user is forwarded back to edit-product.jsp.
• Data access and database update
If validation succeeds, the controller delegates the update operation to the DAO layer.
The DAO constructs and executes a prepared SQL UPDATE statement to apply the changes to the product record.
Prepared statements are used to ensure data integrity and prevent SQL injection.
• Post-processing and redirection
After the database update is completed, control returns to the controller.
The controller redirects the administrator back to the *Products Management* page.
• Response rendering
The product list is reloaded from the database.
The updated product information is displayed in the Products Management interface.
The administrator can continue managing products or navigate to other administrative functions.

2.4.View all orders



Explanation of the code flow when admin clicks on 'orders':

When an administrator views all orders, the system retrieves order data from the database, applies filtering, sorting, and pagination logic, and renders the Orders Management interface.

• Request initiation

The administrator is logged in and currently accessing the Admin Panel (e.g. admin-dashboard.jsp).

The administrator clicks the "Orders" or "View All Orders" option from the admin navigation menu.

The browser sends a GET request to the /orders endpoint, which is mapped to AdminOrdersController.

• Request filtering and authorization

Before reaching the controller, the request is intercepted by the administrative authentication filter.

The filter verifies that:

o A valid user session exists.

o The authenticated user has the admin role.

If either condition fails, the request is rejected and the user is redirected to the login page or shown an access-denied response.

If the validations succeed, the request is forwarded to the controller.

• Controller processing

The AdminOrdersController receives the request and executes its doGet() method.

The controller checks whether any order-related administrative actions (such as status updates) are present in the request.

If no action parameters are detected, the controller proceeds with loading the order list.

• Controller reads filter parameters

The controller retrieves optional filtering inputs from the request, including:

o Order status (e.g. pending, completed, cancelled).

o Date range or keyword (if supported).

If no filter parameters are provided, default values are applied.

• Controller reads sorting parameters

The controller retrieves sorting options from the request:

o sortBy → column used for sorting (e.g. order ID, order date, total amount).

o sortDir → sorting direction (ascending or descending).

Default sorting options are applied if parameters are missing.

• Controller handles pagination logic

The controller reads the current page number from the request.

A fixed page size is applied to limit the number of orders displayed per page.
The database query offset is calculated based on the current page.

• Controller requests total order count from DAO

The controller calls a DAO method to retrieve the total number of orders matching the current filters.
The DAO executes a COUNT(*) query on the orders table.
The total number of matching orders is returned to the controller.

• Controller loads paginated order list from DAO

The controller calls a DAO method to retrieve a paginated list of orders.
The DAO constructs SQL queries incorporating filtering, sorting, and pagination logic.
Prepared statements are used to execute the queries securely.
Each database row is mapped to an Order model object.
A list of orders is returned to the controller.

• DAO returns data to controller

The controller receives:
o A list of orders.
o The total order count for pagination calculations.

• Controller sets request attributes

The controller attaches the retrieved data to the request, including:
o orders
o page
o totalPages
o totalItems
o sortBy / sortDir

• Controller forwards to admin-orders.jsp

The controller forwards the request to admin-orders.jsp using RequestDispatcher.forward().
Request attributes remain available for rendering the view.

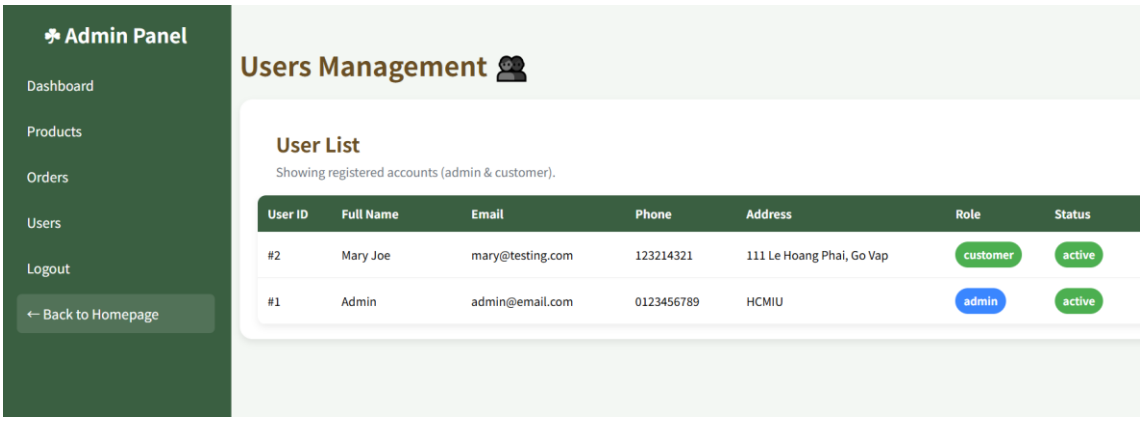
• Response rendering

The admin-orders.jsp page reads the request attributes and renders the Orders Management interface.
The page displays:
o A table of orders (order ID, customer, date, total amount, status).
o Sorting and filtering controls.
o Pagination controls for navigating between pages.

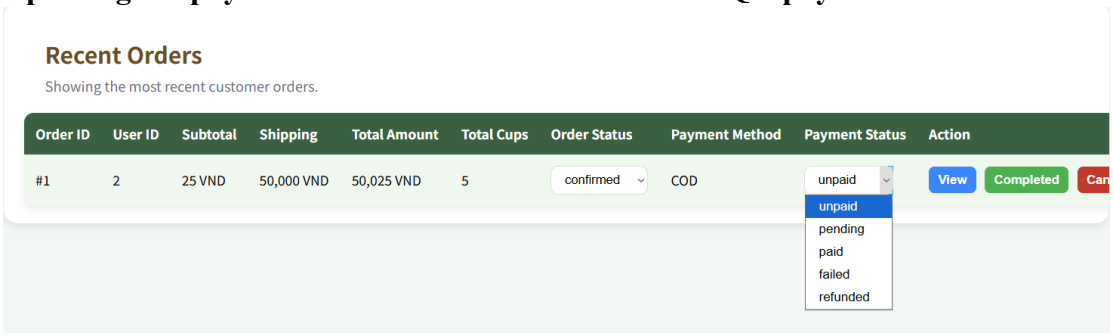
• Final response

The fully rendered Orders Management page is returned to the browser.
The administrator can review all orders or proceed with further order-related actions.

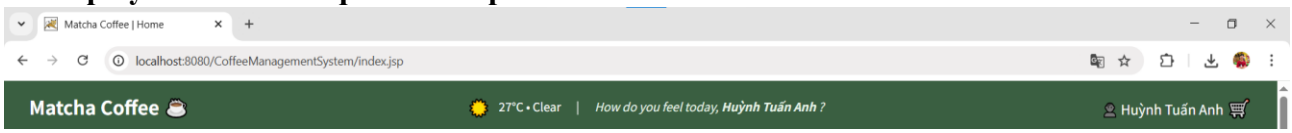
2.5.Get all users' information



2.6.Updating the payment status and order status of VietQR payment



3. Display weather and quote via OpenWeatherAPI



The weather display with greeting message in the navigation bar

Explanation of the code flow of displaying weather information and greeting using OpenWeatherAPI:

– Browser loads the webpage:
+ The navigation bar (navbar) is rendered.
+ The weather container (weather-box) is displayed with default content:
- Weather icon placeholder
- Text "Loading…"
- Greeting message: "How do you feel today, {username}?"
– JSP checks user login status: The JSP uses JSTL <c:choose> to determine whether a user is logged in.
– User greeting logic:
+ If sessionScope.currentUser exists: The user's full name is displayed.
+ If the user is not logged in: The greeting defaults to "Guest".
– Greeting message rendering: The final greeting format is: How do you feel today, <User Name>?. How do you feel today, <User Name>?
– DOMContentLoaded event: Once the page finishes loading, JavaScript triggers the loadWeather() function.
– Check cached weather data
+ The script checks localStorage for cached weather data (weather_hcm).
+ If cached data exists and is less than 30 minutes old: Cached weather data is used and no API request is sent.
– Calling OpenWeather API: If no valid cache exists:

+ JavaScript sends an HTTP request to the OpenWeather API.
+ The request fetches real-time weather data for Ho Chi Minh City.
+ Temperature is retrieved in Celsius.
– Processing API response
+ The response JSON is parsed.
+ Relevant data is extracted from current temperature, and main weather condition (clear, clouds, rain, etc.)
– Caching weather data:
+ Weather data is stored in localStorage with weather details and timestamp
+ This reduces API calls and improves performance.
– Updating weather icon: JavaScript selects a weather icon based on weather condition such ass clear, clouds, rain, snow, default.
– Updating weather text: The temperature and weather condition are displayed in the format: 30°C • Clouds.
– Displaying weather and greeting together: The navigation bar displays weather icon, weather text, separator, and greeting message.
– Handling API failure: If the API request fails:
+ An error is logged to the console.
+ The text “Weather unavailable” is shown instead.

CHAPTER 4: CONCLUSIONS

Conclusion

This project successfully developed a web-based coffee management system that supports both customers and administrators. The system allows users to browse products, manage their shopping cart, place orders, and manage their accounts, while administrators can manage products, orders, and users through the admin panel. All main features work as intended and meet the project requirements.

The system follows the Model–View–Controller (MVC) structure, which helps separate application logic, database access, and user interface components. Controllers handle requests, DAOs manage database operations, and JSP pages display dynamic content. Security is also considered through the use of filters that restrict access to admin functions based on user roles. Features such as pagination, sorting, and filtering improve usability and system performance.

Overall, this project provides a solid foundation for an online coffee store system. It demonstrates the application of core web development concepts, database interaction, and role-based access control in a practical and structured way.

Future Improvements

Although the system meets the basic requirements, there are several areas that can be improved in the future.

First, system security can be enhanced by adding stronger authentication methods, such as multi-factor authentication, and improving password recovery features. Input validation and error handling can also be improved to make the system more robust.

Second, the user interface can be made more user-friendly by improving the layout and design, adding client-side validation, and making the website more responsive on mobile devices. Using a modern front-end framework could also improve the overall user experience.

Third, more features can be added to increase system functionality, such as sales reports, order statistics, inventory tracking, and email notifications for order updates. Integration with online payment gateways and delivery tracking services would make the system more practical for real-world use.

Forth, the order status and payment status can automatically confirm by payment gateway or third-party application.

Finally, the system can be further developed by using modern frameworks such as Spring Boot and building RESTful APIs. This would improve scalability, make the system easier to maintain, and allow it to integrate with other applications in the future.