Team Y

DishIQ
Software Requirements Specification
For DishIQ

Version 2.0

Revision History

Date	Version	Description	Author
18/11/2025	2.0	Introduction	Krista U. Singh
18/11/2025	2.0	All Use Cases	Krista U. Singh
18/11/2025	2.0	E-R Diagram for Entire System	Marina Morcos
18/11/2025	2.0	Detailed Design	Lin Finnegan, Krista U. Singh
18/11/2025	2.0	System Screens	Brianna Hayles, Jing Han Lin
18/11/2025	2.0	Memos of Group Meetings	Marina Morcos
18/11/2025	2.0	Address of the GitHub Repo	Krista U. Singh

Table of Contents

Software Requirements Specification

1. Introduction

1.1 Purpose

The purpose of this Software Requirements Specification (SRS) document is to translate the functional requirements from Version 1 of the SRS into a complete system design. It defines the data structure, class interactions, logic, and interface flow that will guide the implementation. Each diagram and pseudocode block included here provides the blueprint for building the DishIQ application. The focus is on ensuring that every required feature + our creative feature is backed by a clear and consistent design model

1.2 System Overview

DishIQ is an AI-enabled restaurant ordering and management platform that connects customers, managers, chefs, and delivery drivers in one integrated system.

1. Customer/VIP customers can register, browse menus, place and track orders, rate dishes, and deliveries, and chat with an AI assistant.

2. Chef manages menus and marks order progress

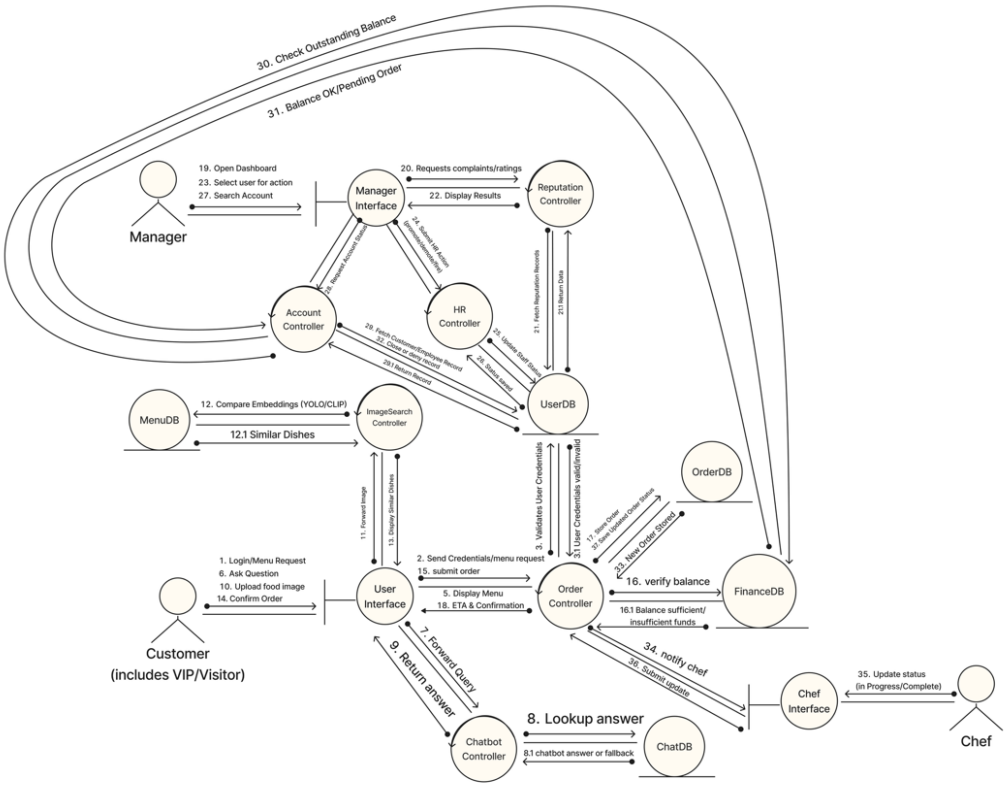
3. Delivery staff bid and fulfill deliveries

4. Managers oversee registration, finances, and complaint handling

5. Visitors can browse and interact with the AI assistant before registering.

On the backend, DishIQ uses Supabase (PostgreSQL) for user, order, and feedback data. Flask API endpoints for logic and authentication. CLIP + YOLO models for the creative image search feature. The frontend (built with Vite + React) handles the GUI that connects each user's role.

1.3 Collaboration Class Diagram



The collaboration diagram shows how all DishIQ actors interact with the system through boundary objects and control classes. Customers, Visitors, and VIP Customers communicate through the **User Interface**, which forwards requests to the **Order Controller**, **Chatbot Controller**, and **Image Search Controller**. These controllers handle login validation, menu retrieval, chatbot responses, image-based dish matching, order placement, and payment verification by accessing shared data stores such as **UserDB**, **MenuDB**, **ChatDB**, **OrderDB**, and **FinanceDB**.

Managers access the system through the **Manager Interface**, which connects to the **Reputation Controller**, **HR Controller**, and **Account Controller** to review complaints, update staff status, and manage account closures. All updates are stored in **UserDB** and **FinanceDB** as needed.

Chefs interact with the system through the **Chef Interface**, receiving order notifications from the **Order Controller,** and sending back preparation updates to **OrderDB**.

Overall, the diagram highlights how DishIQ coordinates user actions, kitchen operations, and managerial tasks through centralized control objects that communicate with shared data stores.

1.4 Design Goals

Our design goals are the following:

- Ensure modular and reusable code across frontend and backend.
- Maintain clear separation of concerns between business logic, data handling, and UI
- Support scalability for multiple restaurants and large data volumes
- Preserve secure data flow using Supabase auth and encrypted transactions
- Seamlessly integrate AI models (LLM Chatbot and CLIP/YOLO) without disrupting the core order flow

2. All Use Cases

2.1 Normal Scenarios & Exceptional Scenarios

The following use cases from Report 1 (UC-01 through UC-12) describe interactions between users and the **DishIQ Web System**. Unless stated otherwise, each use case operates at the **User-Goal level** within the system scope. Each **sequential class diagram** and **Petri Net** encapsulates the normal scenario and the exceptional scenario (alternative flows) within the scope of each individual use case.

UC-01: Visitor Browsing

Primary Actor: Visitor

Goal: To allow visitors to browse a restaurant menu and ask the AI chatbot questions without registering for an account.

Preconditions:

- The system is accessible online
- Menu data is available
- Chatbot is up and running

Main Success Scenario:

1. Visitor opens the homepage.
2. User Interface requests restaurants & search bar from the Browser Controller.
3. Browser Controller requests restaurant list/menu from MenuDB.
   3.1 User Interface sends selection/query to Browser Controller.
   3.2 Browser Controller fetches menu info from MenuDB.
   3.3 MenuDB returns menu results to Browser Controller.
4. Browser Controller displays restaurants & search bar on the User Interface.

5. Visitor asks question (dish or restaurant related).
   5.1 AI ChatBot forwards question to Browser Controller.

6. Browser Controller performs lookup from ChatDB.
   6.1 ChatDB returns answer found.
   6.2 Browser Controller provides chatbot response (answer displayed to Visitor).
   6.3 Display Chatbot Response to User Interface

7. Visitor continues browsing or exits.

Postconditions:

- Visitor can leave the platform or proceed to registration

Alternative Flows:

3a. MenuDB returns error / no menu data.

3a.1 Browser Controller instructs UI to display: "Menu failed to load."

6a. ChatDB returns no usable answer.

6a.1 Browser Controller instructs ChatBot to suggest contacting support.
6a.2 ChatBot displays: "Unable to answer. Please contact support."

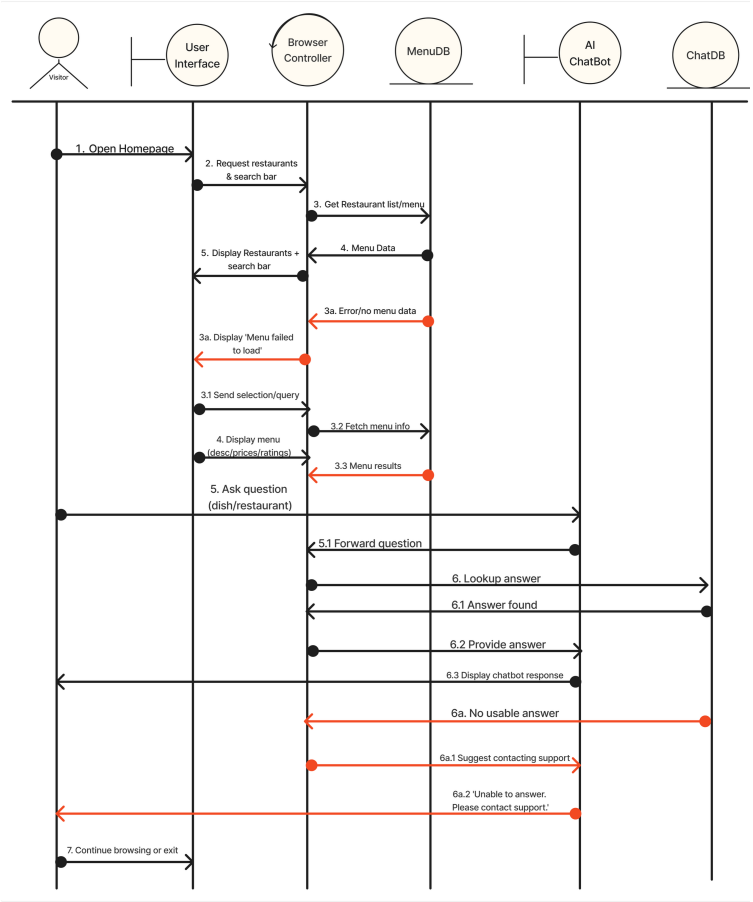


Figure 2.1 – DishIQ's UC-01 Sequential Class Diagram

UC-02: Customer Registration

Primary Actor: Visitor

Goal: To create a customer account and gain access to full ordering features.

Main Success Scenario:

1. Visitor clicks "Register."

2. User Interface requests the registration form from the RegistrationController.
   2.1 RegistrationController displays the registration form back to the User Interface.

3. Visitor enters name, email, password, and payment information.
   3.1 User Interface submits registration data to the RegistrationController.

4. RegistrationController validates the input and checks if the email already exists by sending a request to UserDB.
   4.1 UserDB returns "Information valid & stored."

5. RegistrationController sends a confirmation message to the Email System.
   5.1 Email System sends confirmation email/message to the Visitor.

6. RegistrationController notifies the User Interface that registration succeeded, and the user may now log in.

Postconditions: New customer account is created

Alternative Flows:

4a. UserDB returns "Invalid data."
4a.1 RegistrationController instructs the User Interface to prompt the visitor to correct their input.

5a. UserDB returns "Email exists."
5a.1 RegistrationController instructs the User Interface to display: "Account already registered."

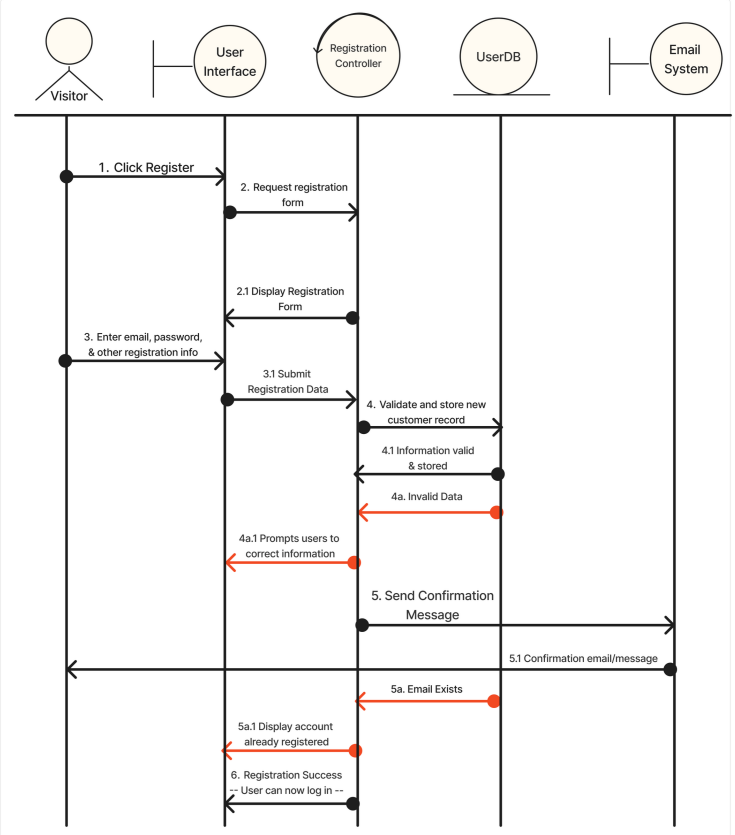


Figure 2.2 – DishIQ's UC-02 Sequential Class Diagram

UC-03: Place Order

Primary Actor: Customer/VIP Customer

Goal: To select dishes, submit payment, and place an order.

Main Success Scenario:

1. Customer logs in
2. System displays menu
3. Customer adds menu items to their cart
4. Customer reviews and confirms their order
5. System verifies necessary funds
6. System sends the order to the kitchen
7. System provides customer with an estimated delivery time
8. System checks deposit balance via Finance Module

Postconditions: Order is recorded and waiting to be prepared

Alternative Flows:

- **5a.** Insufficient balance ---> Prompt to add funds
- **6a.** Kitchen offline ---> System saves the order in the queue to be submitted later

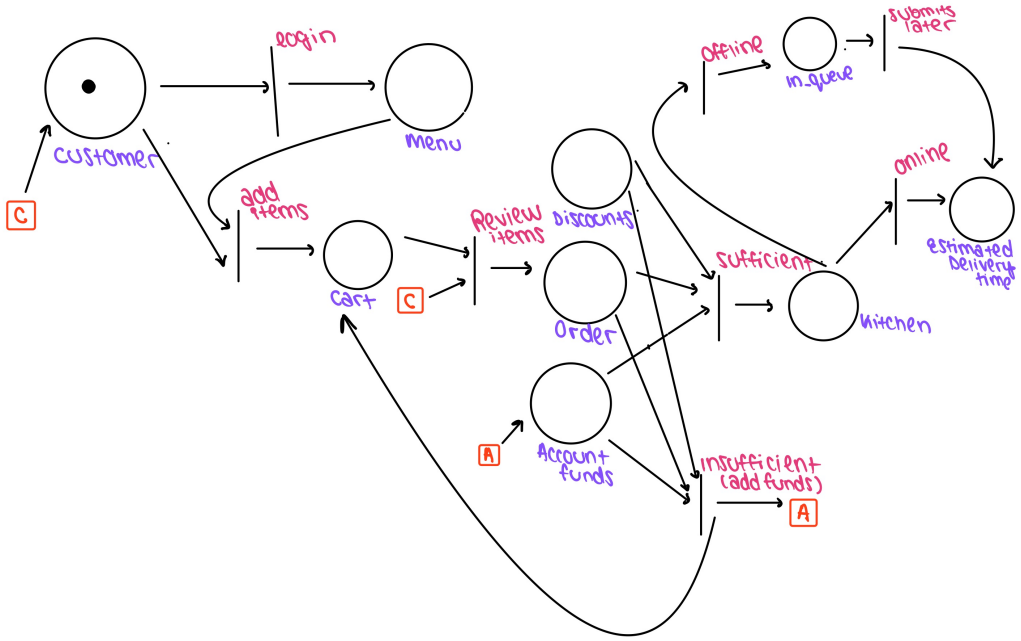


Figure 2.3 – DishIQ's UC-03 Petri Net Diagram

UC-04: Kitchen Receives Order

Primary Actor: Chef

Goal: To receive, prepare, and update the status of the order

Main Success Scenario:

1. OrderDB sends a new order notification to the Kitchen Controller.
   1.1 Kitchen Controller displays the order notification to the Chef through the Chef Interface.

2. Chef views dish details and customer notes using the Chef Interface.
   2.1 Chef Interface requests order details from the Kitchen Controller.
   2.2 Kitchen Controller requests full order details from OrderDB.
   2.3 OrderDB returns dish details and notes to the Kitchen Controller.
   2.4 Kitchen Controller displays the dish details to the Chef.

3. Chef sets the order status to "In Progress."
   3.1 Chef Interface sends status update to the Kitchen Controller.
   3.2 Kitchen Controller saves the status "In Progress" to OrderDB.

4. Chef prepares the dish.
   (No system interaction—physical kitchen action.)

5. Chef sets the order status to "Complete."
   5.1 Chef Interface sends completion status to the Kitchen Controller.
   5.2 Kitchen Controller saves the status "Complete" to OrderDB.

6. Kitchen Controller notifies the Delivery System that the order is Ready for Pickup**.**

Postconditions: The order is ready to be picked up by the delivery person

Alternative Flows:

2a. Chef flags an ingredient issue via the Chef Interface.

2a.1 Chef Interface reports the issue to the Kitchen Controller.

2a.2 Kitchen Controller notifies the Manager System for resolution.

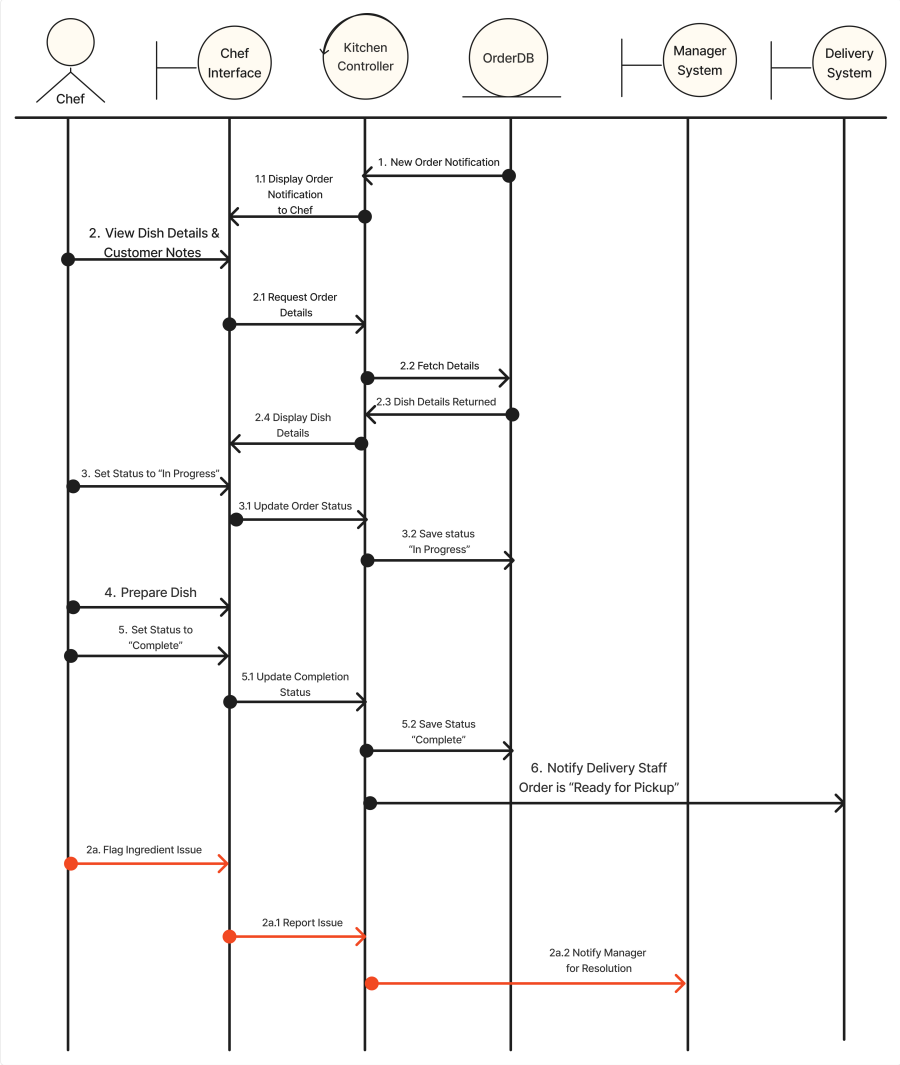


Figure 2.4 – DishIQ's UC-04 Sequential Class Diagram

UC-05: Delivery Person Receives Order

Primary Actor: Delivery Person

Goal: To pick up the order from the restaurant and deliver it to the customer

Main Success Scenario:

1. Delivery Person views available orders through the Delivery Interface.
   1.1 Delivery Interface sends a request for available orders to the DeliveryController.
   1.2 DeliveryController requests the order list from OrderDB.
   1.3 OrderDB returns the list of available orders to the DeliveryController.
   1.4 DeliveryController displays the available orders on the Delivery Interface.

2. DeliveryController assigns an order to the Delivery Person based on bids or availability.
   2.1 Delivery Interface displays the assigned order details to the Delivery Person.

3. Delivery Person sets the delivery status to “In Progress.”
   3.1 Delivery Interface sends the updated status to the DeliveryController.
   3.2 DeliveryController saves the status “In Progress” to OrderDB.

4. Delivery Person sets the status to “Delivered.”
   4.1 Delivery Interface sends completion update to the DeliveryController.
   4.2 DeliveryController saves the status “Delivered” in OrderDB.

5. DeliveryController notifies the Customer System that the order has been delivered and a rating is requested.

Postconditions: Delivery marked complete, and rating recorded.

Alternative Flows:

3a. DeliveryController notifies the Customer System of the delay

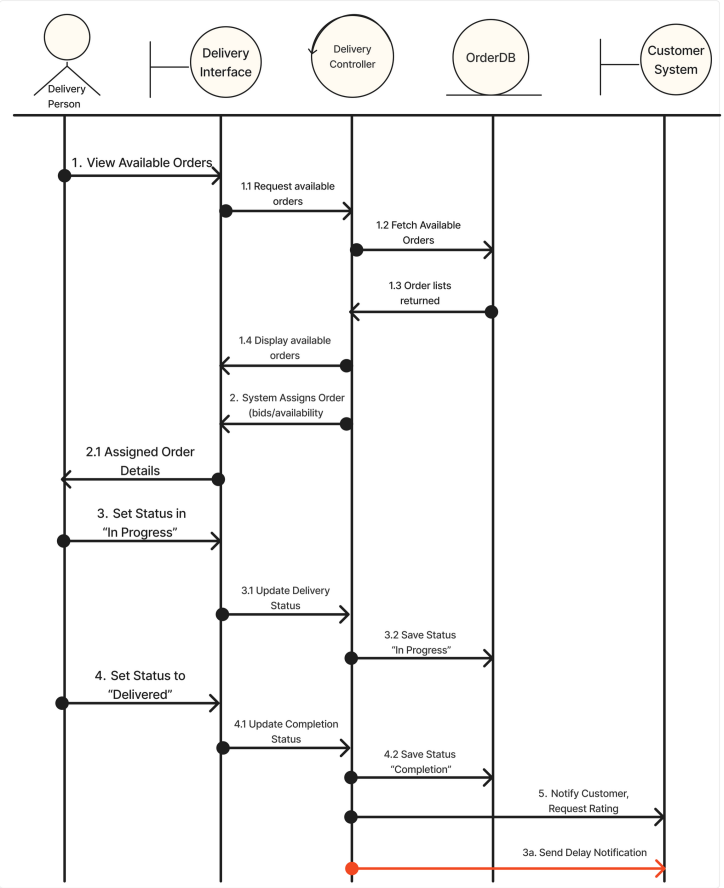


Figure 2.5 – DishIQ’s UC-05 Sequential Class Diagram

UC-06: Customer Rating and Feedback

Primary Actor: Customer

Goal: To rate their food and delivery service after receiving their order

Main Success Scenario:

1. Customer logs in after delivery
2. System shows orders awaiting feedback
3. Customer rates food
4. Customer rates delivery
5. Customer adds any positive or negative comments
6. System stores feedback and notifies manager

Postconditions: Feedback is saved for manager of restaurant to review

Alternative Flows:

3a. Customer skips feedback ---> System reminds them later

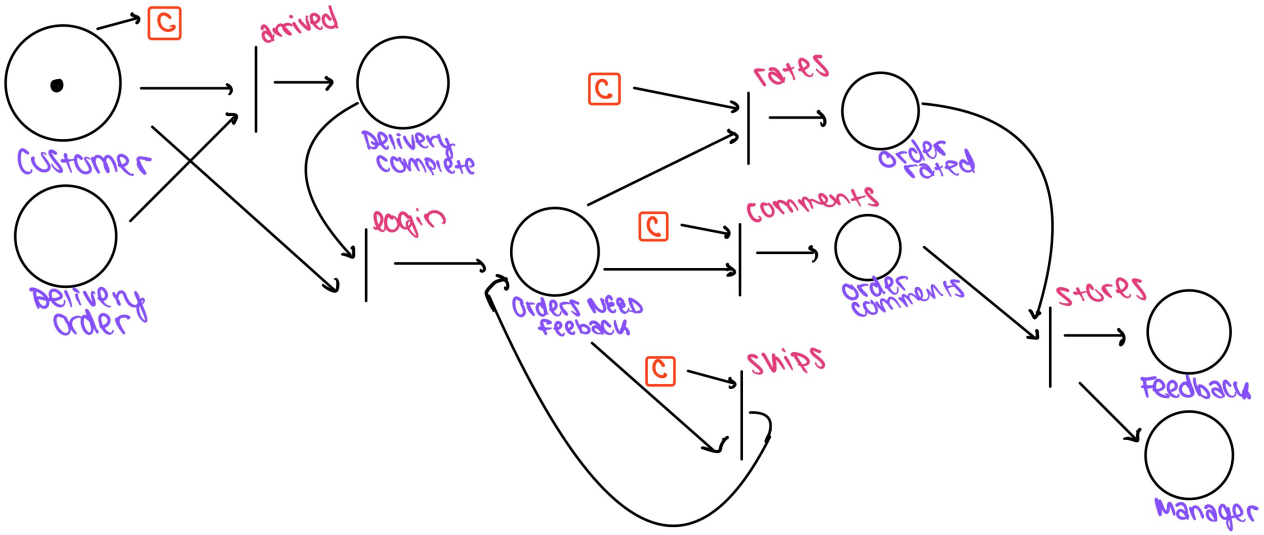


Figure 2.6 – DishIQ's UC-06 Sequential Class Diagram

UC-07: Reputation Handling

Primary Actor: Manager

Goal: To process customer reviews

Main Success Scenario:

1. Manager logs into Admin Dashboard.
   1.1 Manager Interface sends login request to ReputationController.
   1.2 ReputationController verifies manager credentials in UserDB.
   1.3 UserDB returns to ReputationController "Credentials Verified"
   1.4 ReputationController returns dashboard access to Manager Interface.

2. Manager Interface requests recent complaints/compliments.
   2.1 ReputationController queries ReviewDB for recent entries.
   2.2 ReviewDB returns complaints and compliments.
   2.3 Manager Interface displays them to the manager.

3. Manager selects a review to examine.
   3.1 Manager Interface requests review details from ReputationController.
   3.2 ReputationController fetches full review details from ReviewDB.
   3.3 ReviewDB returns review details to ReputationController
   3.4 Manager Interface displays detailed review

4. Manager chooses to issue a warning or commendation.
   4.1 Manager Interface submits action to ReputationController.
   4.2 ReputationController updates staff reputation record in UserDB.

5. System updates the reputation record.
   5.1 ReputationController confirms update to Manager Interface.

6. If complaint is upheld: ReputationController issues warning in UserDB & Notification system notifies the staff member.

Postconditions: Reputation database updated, and user notified.

Alternative Flows:

4a. Manager marks complaint as invalid.
4a.1 Manager Interface sends dismissal action to ReputationController.
4a.2 ReputationController marks complaint as dismissed in ReviewDB.
4a.3 Dismissal is saved in ReviewDB and returns to ReputationContoller

4a.4 Notification System notifies customer that complaint is dismissed.

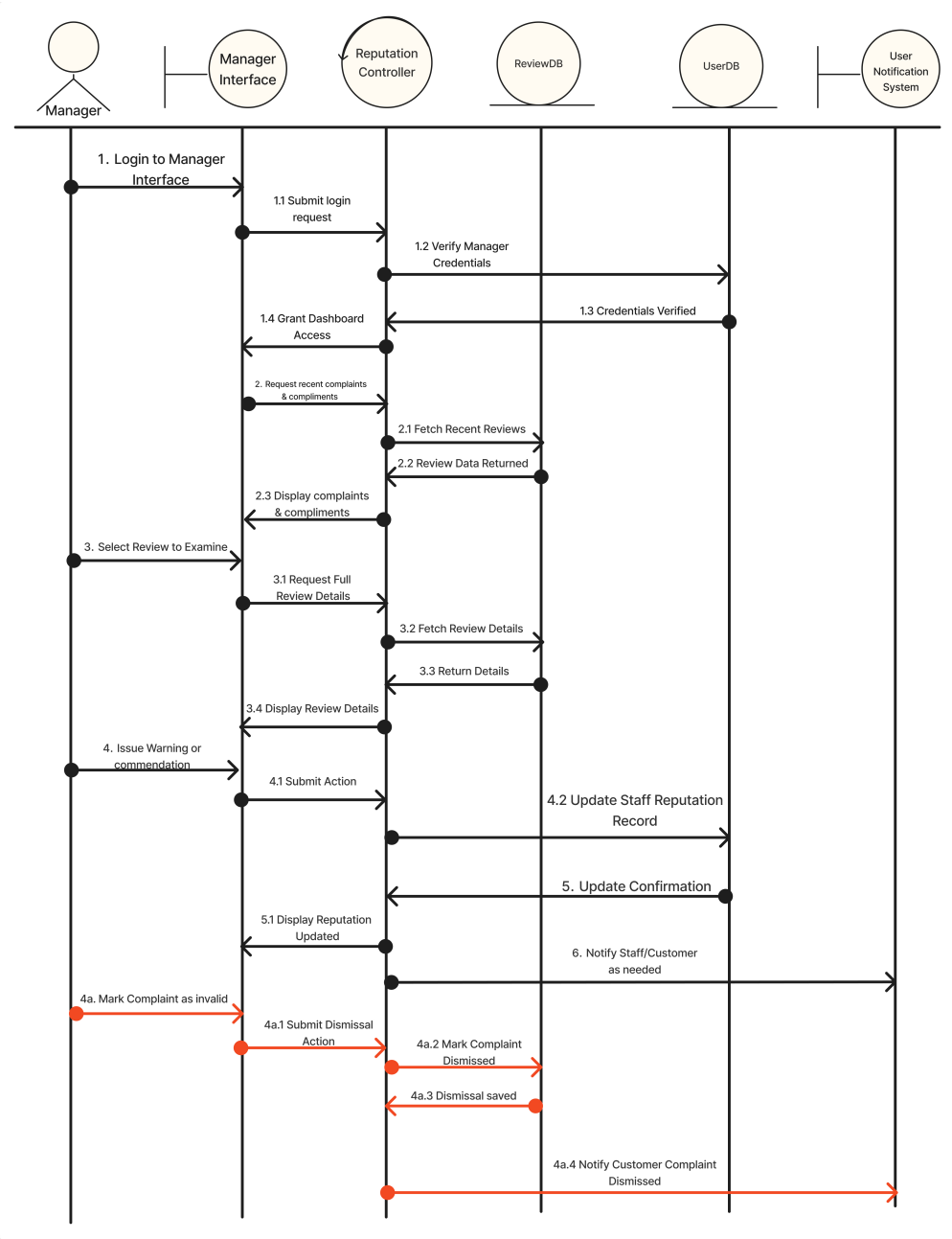


Figure 2.7 – DishIQ's UC-07 Sequential Class Diagram

UC-08: HR Actions

Primary Actor: Manager

Goal: To promote, demote, or terminate staff based on performance.

Main Success Scenario:

1. Manager opens the staff performance dashboard.
   - **1.1** Staff Dashboard sends a request to HRController for performance data.
   - **1.2** HRController requests warning counts and ratings from StaffDB.
   - **1.3** StaffDB returns performance data to HRController.

2. HRController displays the warning counts and ratings on the Staff Dashboard.

3. Manager selects a staff member and chooses an action (promote, demote, or fire).
   3.1 Staff Dashboard submits the HR action to HRController.

4. HRController updates the employee status in StaffDB.
   4.1 StaffDB confirms update saved and returns confirmation to HRController.

5. HRController notifies the affected staff member through the Notification System.
   5.1 Staff Dashboard displays confirmation that HR records have been updated.

Postconditions: HR records updated and visible in manager reports.

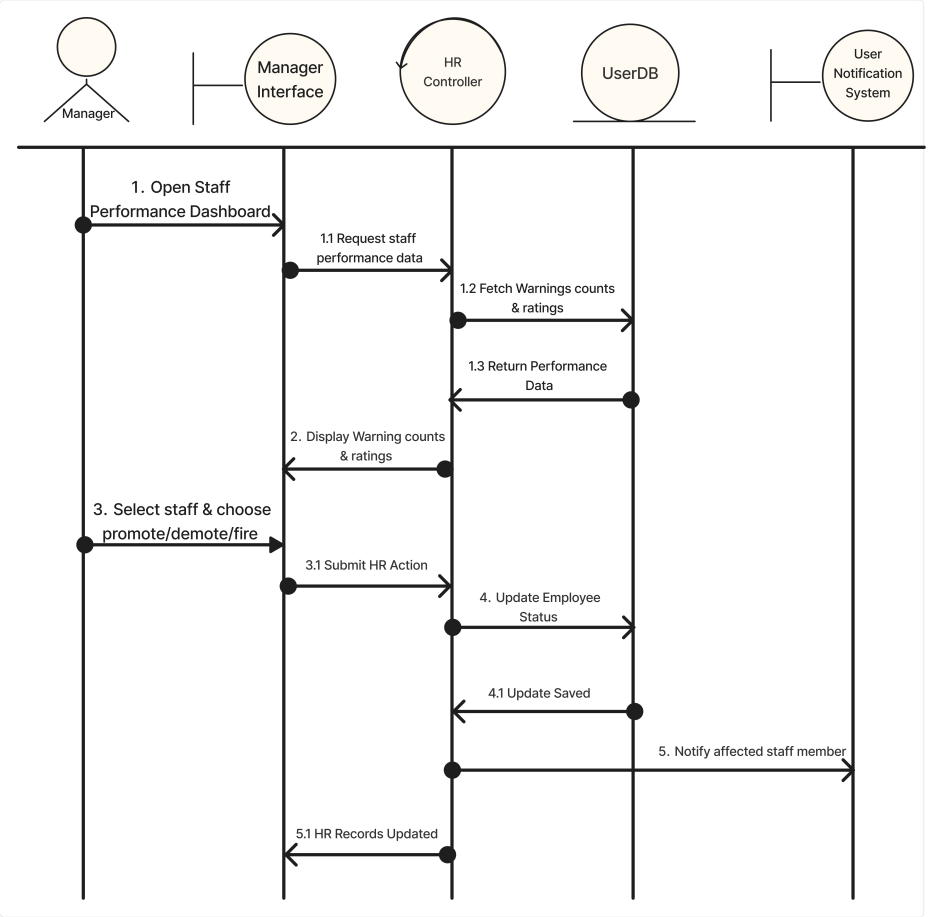


Figure 2.8 – DishIQ's UC-08 Sequential Class Diagram

UC-09: Deposit and Finance Management

Primary Actor: Customer

Goal: To verify sufficient funds before orders are processed.

Main Success Scenario:

1. Customer initiates payment for the order.
   1.1 Payment Interface submits a payment request to the FinanceController.

2. FinanceController checks the customer's deposit balance in FinanceDB.
   2.1 FinanceDB returns "Balance sufficient."

3. FinanceController deducts the order amount and logs the transaction in FinanceDB.
   3.1 FinanceDB confirms the transaction has been saved.

4. FinanceController updates the order as paid in OrderDB.
   4.1 OrderDB confirms the order record has been updated.

4.2 FinanceController sends a “Payment Successful” confirmation to the Payment Interface.

Postconditions: Order and financial record updated

Alternative Flows:

2a. Insufficient Funds - FinanceDB returns “Balance too low.”

2a.1 FinanceController prompts the customer via the Payment Interface to add funds before continuing.

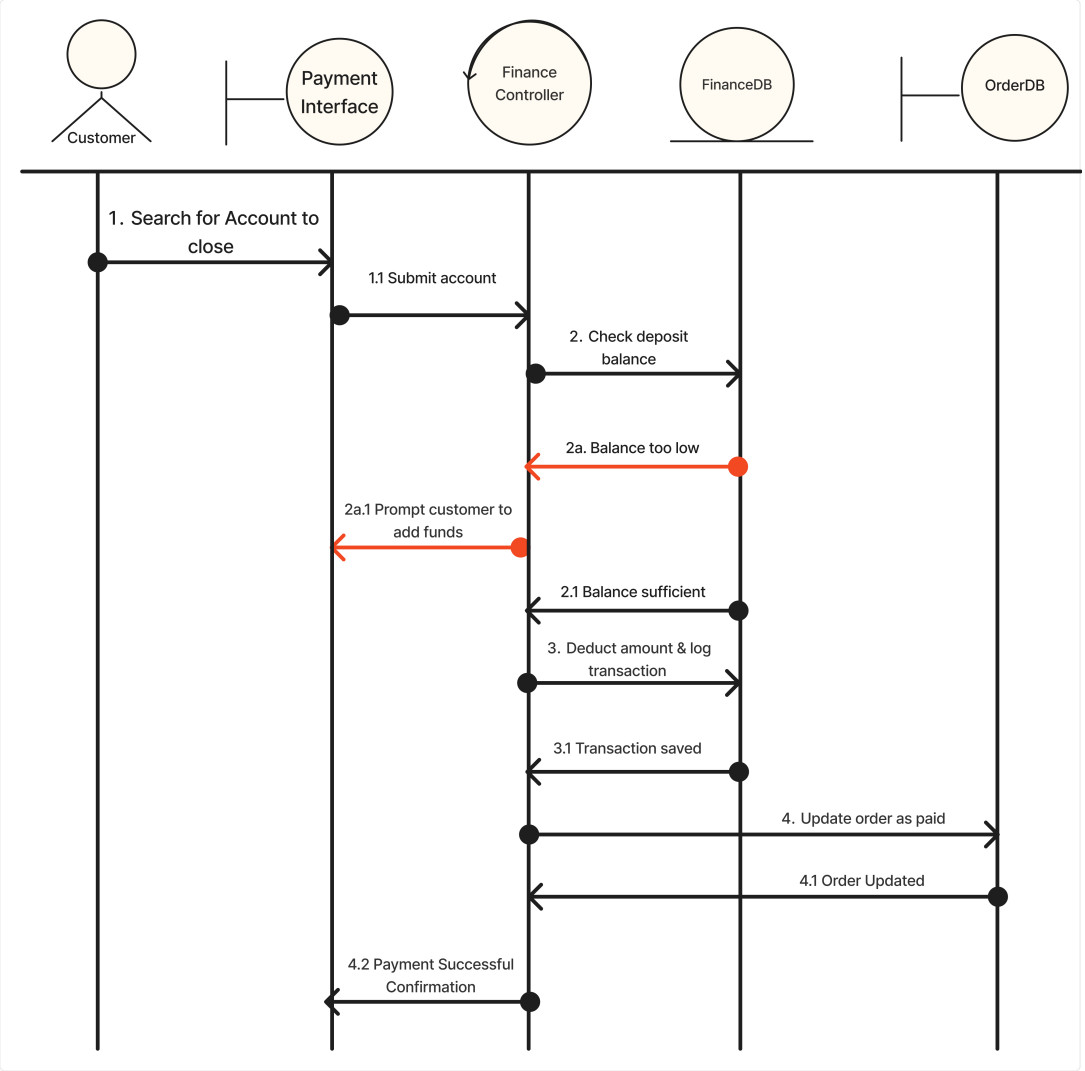


Figure 2.9 – DishIQ’s UC-09 Sequential Class Diagram

UC-10: Account Closure

Primary Actor: Manager

Goal: To close a customer or employee account

Main Success Scenario:

1. Manager searches for the account to close using the Admin Dashboard.
   - **1.1** Admin Dashboard submits the account search query to the AccountController.
   - **1.2** AccountController retrieves account info from UserDB.
   - **1.3** UserDB returns account details to the AccountController.
   - **1.4** AccountController displays the account details on the Admin Dashboard.

2. Admin Dashboard requests status check for outstanding balances and orders.
   - **2.1** AccountController checks for pending orders in OrderDB.
   - **2.2** OrderDB returns pending order status to AccountController.

2.3 AccountController checks outstanding balances in FinanceDB.
2.4 FinanceDB returns balance status (clear or settled).
2.5 AccountController indicates the account is ready for closure on the Admin Dashboard.

3. Manager confirms closure of the account.
   3.1 Admin Dashboard submits the closure confirmation to AccountController.

4. AccountController clears deposits through FinanceDB.
   4.1 FinanceDB confirms deposits cleared.
   4.2 AccountController blacklists and deactivates the user account in UserDB.
   4.3 UserDB confirms the account is deactivated.
   4.4 AccountController displays successful closure confirmation on the Admin Dashboard.

Postconditions: Account removed from active records.

Alternative Flows:

2a. If OrderDB indicates a pending order, AccountController prevents closure and displays: "Closure prevented — pending order detected."

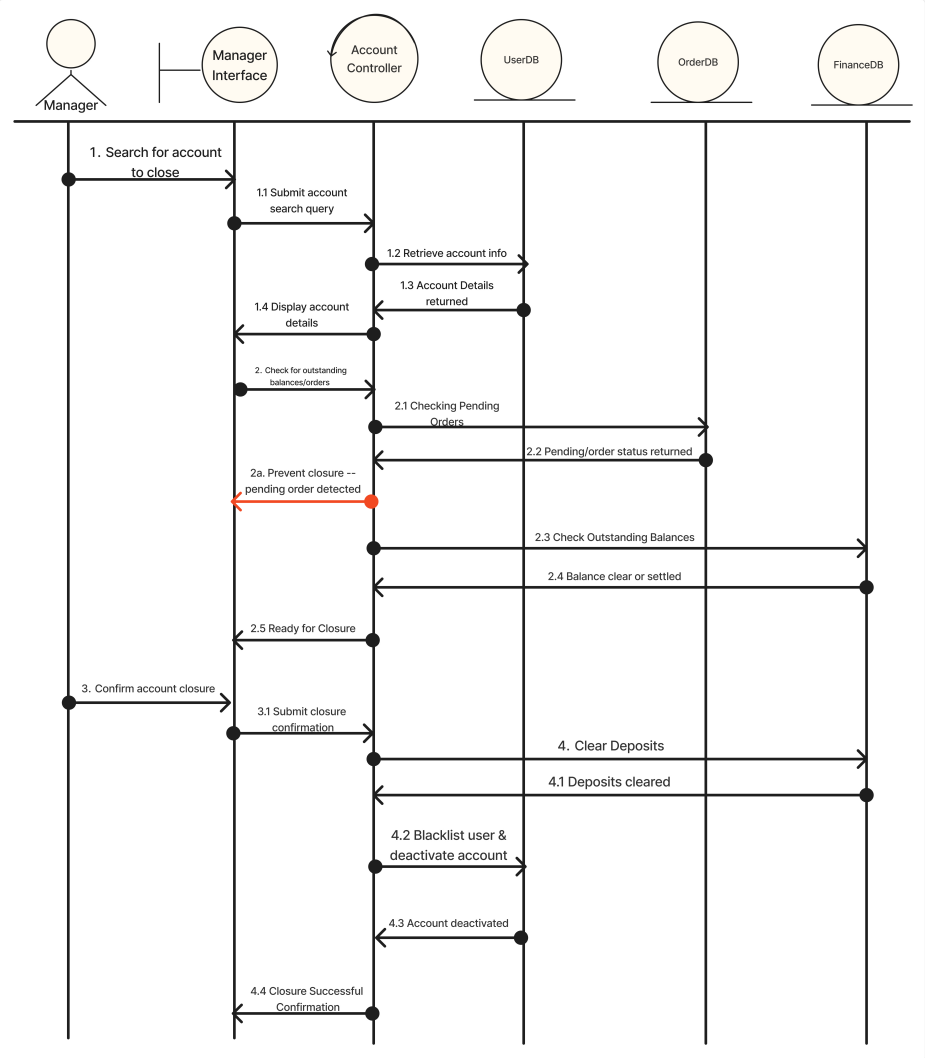


Figure 2.10 – DishIQ's UC-10 Sequential Class Diagram

UC-11: AI Chatbot Interaction

Primary Actor: Visitor, Customer, VIP Customer

Goal: To interact with the AI Chatbot for assistance

Main Success Scenario:

1. User open chatbot interface
2. User asks a question
3. System searches local knowledge base
4. Chatbot answers questions if relevant data found
5. If the answer is not found, the system queries the external LLM
6. Answer is given to user

Postconditions: User receives relevant information

Alternative Flows:

5a. LLM unavailable ---> System falls back to the knowledge base---> System returns a fallback message.

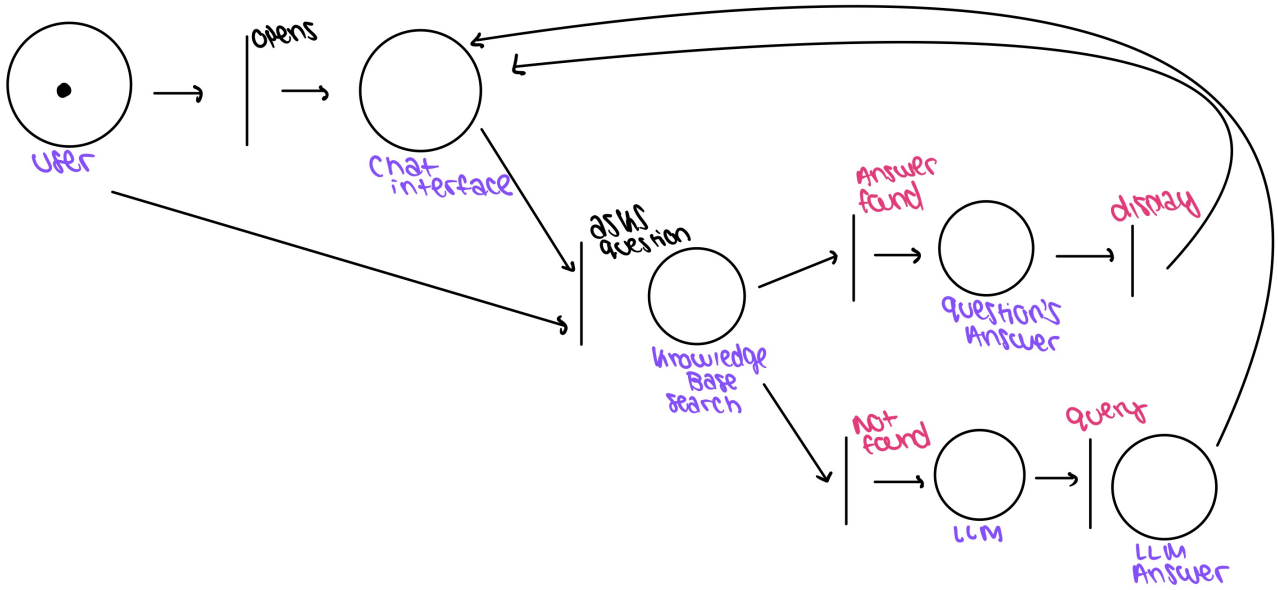


Figure 2.11 – DishIQ's UC-11 Petri Net Diagram

UC-12: Creative Feature

Primary Actor: Customer, Visitor (User)

Goal: To allow users to upload a photo of food and receive visually similar menu suggestions.

Main Success Scenario:

1. User opens the Image Search Tool.
   1.1 Image Search Interface sends initialization request to the ImageAnalysisController.
   1.2 ImageAnalysisController displays upload options on the Image Search Interface.

2. User uploads or captures a food image.
   2.1 Image Search Interface submits the uploaded image to the ImageAnalysisController.

3. ImageAnalysisController sends the image to the YOLO Model for object detection.
   3.1 YOLO Model returns detected objects and bounding boxes.
   3.2 ImageAnalysisController sends the image to the CLIP Model to generate an embedding.
   3.3 CLIP Model returns the embedding vector.

4. ImageAnalysisController compares the generated embedding against stored embeddings in MenuDB.
   4.1 MenuDB returns matching dish embeddings.

5. Image Search Interface displays similar menu items (names, ratings, prices).

6. User clicks a suggested dish to view details or begin an order.
   6.1 Image Search Interface requests detailed dish view from the ImageAnalysisController.

Postconditions:

- Matching dishes are displayed.
- User interaction is logged for personalization.

Alternative Flows:

3a. Image Search Interface reports upload failure - user is prompted to retry the upload.

4a. Image Search Interface displays “No similar dishes found.”

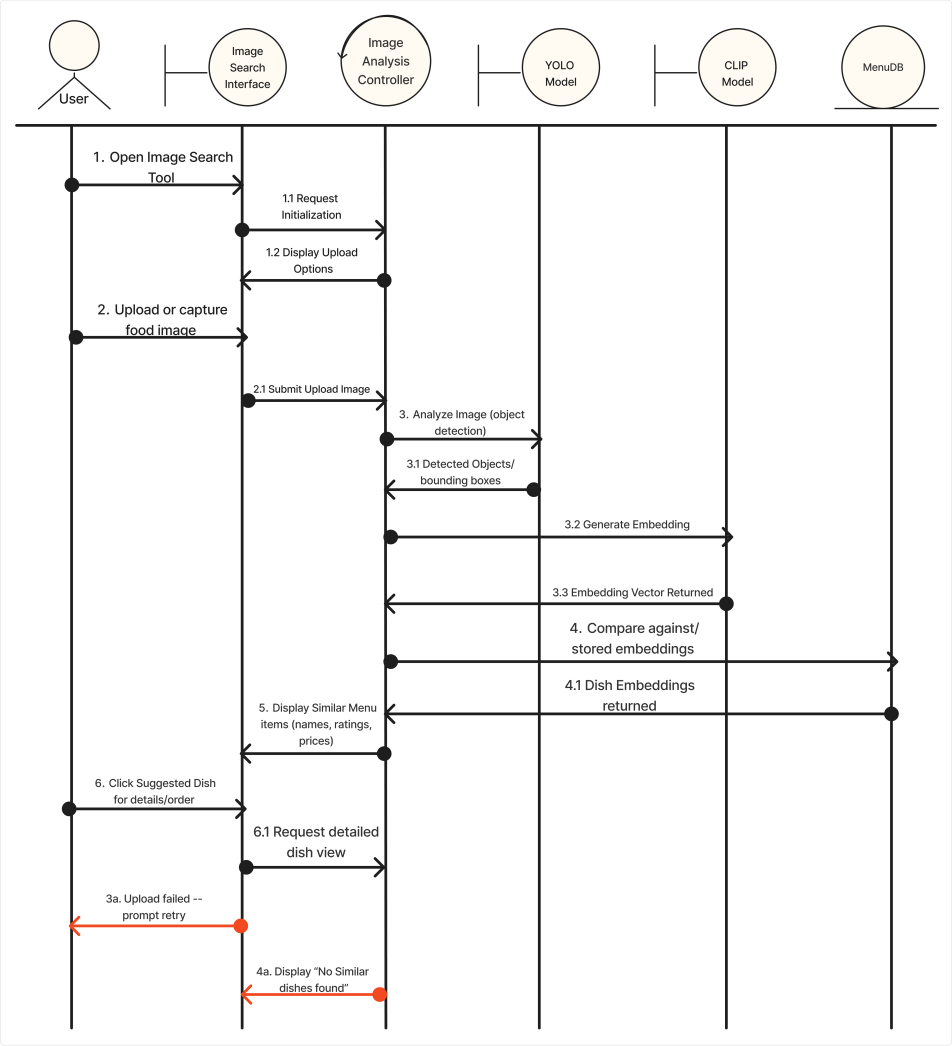


Figure 2.12 – DishIQ’s UC-12 Sequential Class Diagram

3. E-R Diagram

3.1 Data Modeling and Entity-Relation (ER) Diagram

DishIQ uses Supabase (PostgreSQL) to store data relations within the system constraints. Below displays our data modeling diagram given via Supabase platform and the translation of it into an Entity-Relation (ER) Diagram for our purposes of this report.

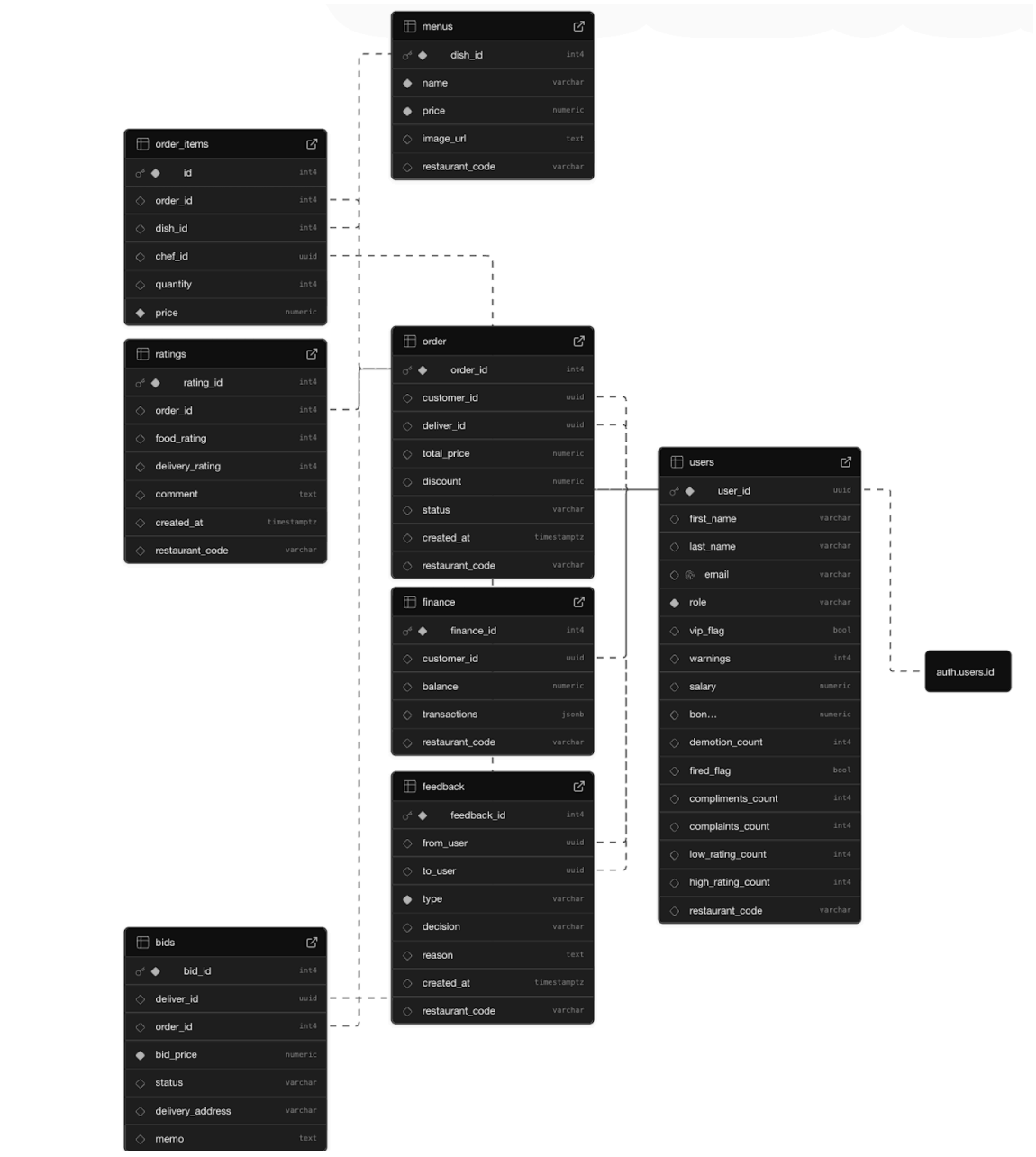


Figure 3.1 – DishIQ's Data Modeling Diagram

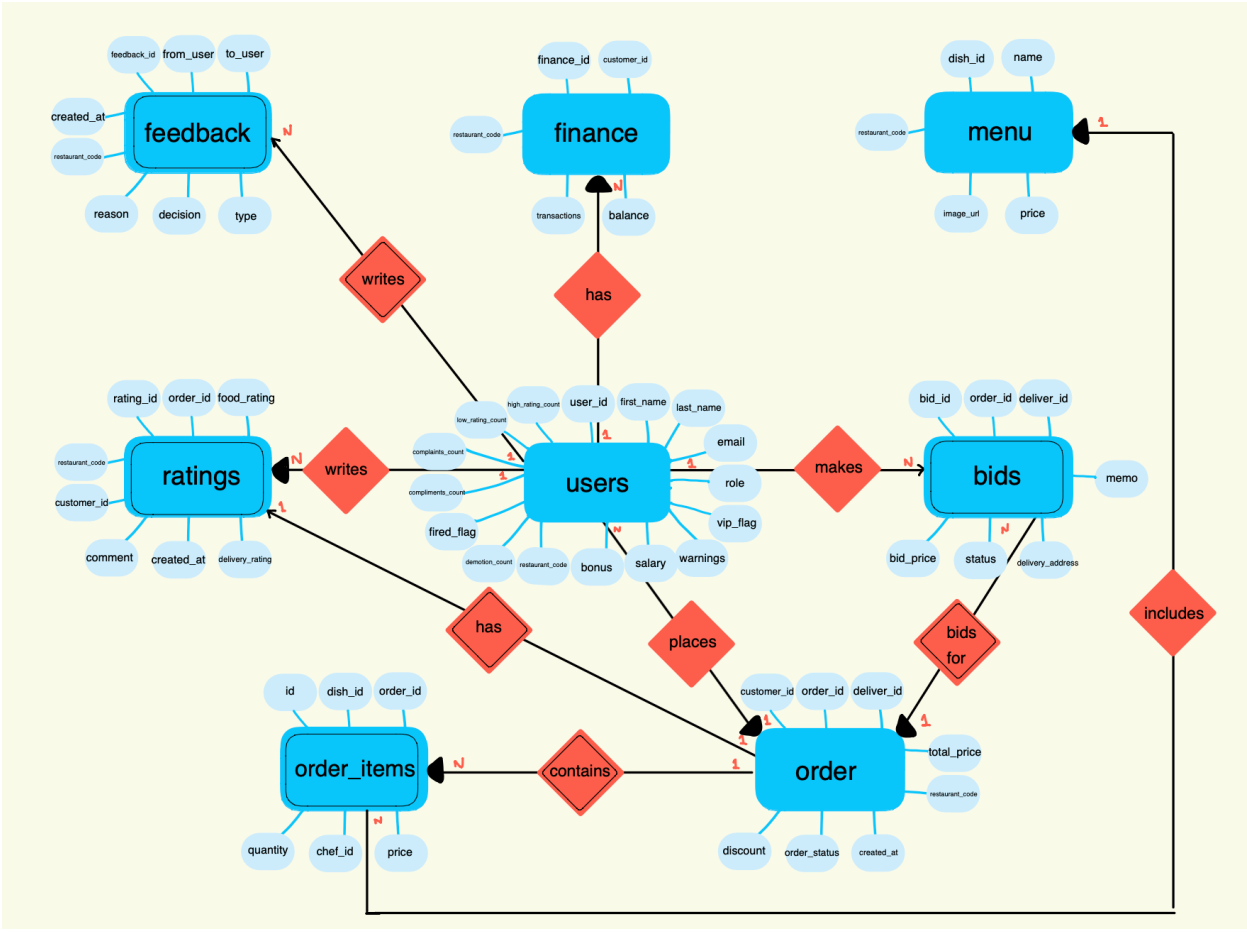


Figure 3.2 – DishIQ's Entity Relationship Diagram

The entity relationship diagram captures how the database of the Dish IQ platform is connected and the information shared in each perspective table. In the database, there are eight tables storing various forms of data. The users table stores all the DishIQ users' information from customers to employees and their related attributes. Some examples of attributes if a customer is VIP, or an employee's salary. Attributes that are not applicable to a certain user are not applied and left NULL. The finance table stores a customer's financial account with information such as current balance and transaction history. The order table represents each customer's order placed on the platform. It includes details such as order status and total price. Since each order may contain multiple items and list of dish ids are not possible, the database models this through the order_items table. In this table the dish price, quantity and the order its related to is stored. The order_item table is also connected to the menu table which stores dish name, price and image. The bids table handles the delivery bidding system. Delivery drivers submit bids on orders, and each bid includes fields such as bid price, delivery address and status of the bid decision made by the manager. The ratings and feedback tables store user-generated evaluations. The ratings table keeps track of ratings from a user's order from a scale of 1-5 including food rating and delivery rating. The feedback table records internal feedback exchanged between users, such as compliments, complaints, or decisions made by the manager. Overall, the diagram illustrates how DishIQ manages customers, employees, orders, deliveries, finances, feedback, and menu items in an interconnected system. Each entity is linked through clearly defined relationships, ensuring that data flows seamlessly across all components of the platform.

4. Detailed Design

4.1 Pseudocode for System Functionalities

Below are each of the features of DishIQ with its corresponding pseudocode on the implementation of it via the backend of the application.

```
# ============================================================
# UC-01: Visitor Browsing - Visitor Menu Browsing
# ============================================================
# ------------------------------------------------------------
# UC-01 - Browse Menu Items as Visitor
# Purpose: Allow non-registered visitors to browse available menu items without seeing VIP-exclusive items
# Input:
#     - visitor: Visitor object (non-registered user)
#     - menu_items: List[MenuItem] (all available menu items in system)
# Output:
#     - visible_items: List[Dict] (menu items visible to visitors, excluding early access items)

FUNCTION browse_menu(visitor, menu_items):
    # Initialize empty list for visible items
    visible_items = []

    # Log the browsing action
    LOG("Visitor {visitor.user_id} is browsing menu")

    # Iterate through all menu items
    FOR each item IN menu_items:
        # Check if item is NOT early access (VIP-only)
        IF item.is_early_access == False:
            # Convert item to dictionary format
            item_dict = item.to_dict()
            # Add to visible items list
            visible_items.append(item_dict)

    # Return filtered list of visible items
    RETURN visible_items
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-01 - Visitor Permission Check
# Purpose: Verify what actions a visitor can perform
# Input:
#     - visitor: Visitor object
#     - action: String (action name to check permission for)
# Output:
#     - has_permission: Boolean (True if visitor can perform action, False otherwise)

FUNCTION check_permissions_visitor(visitor, action):
    # Define allowed actions for visitors
    allowed_actions = ["browse_menu", "apply_for_registration"]

    # Check if requested action is in allowed list
    IF action IN allowed_actions:
        RETURN True
    ELSE:
        RETURN False
END FUNCTION
```

```
# UC-02 - Apply for Registration as Visitor
# Purpose: Allow visitors to apply for customer registration and automatically check blacklist
# Input:
#     - visitor: Visitor object
#     - full_name: String (applicant's full name)
#     - phone: String (applicant's phone number)
#     - address: String (delivery address)
#     - password: String (account password)
#     - blacklist: List[String] (list of blacklisted email addresses)
# Output:
#     - registration_application: Dict (application data) OR Exception
# Exceptions:
#     - RegistrationException: If application already submitted
#     - BlacklistedUserException: If email is blacklisted

FUNCTION apply_for_registration(visitor, full_name, phone, address, password, blacklist):
    # Check if visitor has already submitted an application
    IF visitor.registration_application IS NOT None:
        THROW RegistrationException("Registration application already submitted")

    # Check if blacklist checking is enabled
    IF BLACKLIST_ENABLED == True:
        # Check if visitor's email is in blacklist
        IF visitor.email IN blacklist:
            LOG("Blacklisted registration attempt: {visitor.email}")
            THROW BlacklistedUserException("This email is blacklisted")

    # Hash the password for security
    password_hash = SHA256_HASH(password)

    # Create registration application dictionary
    application = {
        "applicant_id": visitor.user_id,
        "username": visitor.username,
        "email": visitor.email,
        "full_name": full_name,
        "phone": phone,
        "address": address,
        "password_hash": password_hash,
        "application_date": CURRENT_TIMESTAMP(),
        "status": "pending"
    }

    # Store application in visitor object
    visitor.registration_application = application

    # Log successful submission
    LOG("Visitor {visitor.user_id} submitted registration application")

    # Return copy of application
    RETURN application.copy()
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-02 - Manager Approves Registration
# Purpose: Manager reviews and approves customer registration applications
# Input:
#     - manager: Manager object
#     - application: Dict (registration application data)
# Output:
#     - customer: Customer object (newly created customer account)

FUNCTION approve_registration(manager, application):
    # Verify manager has permission
    IF NOT manager.has_permission("approve_registration"):
        THROW UnauthorizedAccessException("Manager lacks permission")

    # Extract application data
    user_id = GENERATE_UNIQUE_ID("C-")
    username = application["username"]
    email = application["email"]
    full_name = application["full_name"]
    phone = application["phone"]
    address = application["address"]
    password_hash = application["password_hash"]

    # Create new Customer object
    customer = CREATE Customer(
        user_id=user_id,
        username=username,
        email=email,
        full_name=full_name,
        phone=phone,
        address=address,
        password_hash=password_hash
    )

    # Log approval
    LOG("Manager {manager.user_id} approved registration for {email}")

    # Return new customer
    RETURN customer
END FUNCTION
```

```
# ------------------------------------------------------
# UC-03 - VIP Customer Places Order with Benefits
# Purpose: Allow VIP customers to place orders with 5% discount and free delivery every 3rd order
# Input:
#     - vip_customer: VIPCustomer object
#     - menu_items: List[MenuItem] (items to order)
#     - quantities: List[Integer] (quantity for each item)
#     - delivery_fee: Float (default 5.0, but may be 0 for 3rd order)
# Output:
#     - order: Order object (created order with discounts applied)
# Exceptions:
#     - InsufficientFundsException: If balance too low

FUNCTION place_order_vip_customer(vip_customer, menu_items, quantities, delivery_fee=5.0):
    # Calculate order count (including this order)
    order_count = LENGTH(vip_customer.order_history) + 1

    # Check if this is 3rd, 6th, 9th order (free delivery)
    IF order_count MOD 3 == 0:
        delivery_fee = 0.0  # Free delivery
    ELSE:
        delivery_fee = 5.0  # Regular delivery fee

    # VIP discount rate (5%)
    discount_rate = 0.05

    # Calculate subtotal
    subtotal = 0.0
    FOR i FROM 0 TO LENGTH(menu_items) - 1:
        item = menu_items[i]
        quantity = quantities[i]
        subtotal = subtotal + (item.price * quantity)

    # Calculate discount amount
    discount_amount = subtotal * discount_rate

    # Generate unique order ID
    order_count = LENGTH(vip_customer.order_history) + 1
    timestamp = CURRENT_TIMESTAMP()
    order_id = "ORD-{vip_customer.user_id}-{order_count}-{timestamp}"

    # Create order items list
    order_items = []
    FOR i FROM 0 TO LENGTH(menu_items) - 1:
        item = menu_items[i]
        quantity = quantities[i]

        order_item = CREATE OrderItem(
            menu_item_id=item.item_id,
            name=item.name,
            price=item.price,
            quantity=quantity
        )
        order_items.append(order_item)

    # Create Order object
    order = CREATE Order(
        order_id=order_id,
        customer_id=vip_customer.user_id,
        items=order_items,
        delivery_fee=delivery_fee,
        discount=discount_amount,
        delivery_address=vip_customer.address
    )

    # Check if VIP customer has sufficient funds
    IF vip_customer.account_balance < order.total_amount:
        # Automatic warning for insufficient funds
        vip_customer.warnings = vip_customer.warnings + 1
        LOG("VIPCustomer {vip_customer.user_id} received warning for insufficient funds")

        THROW InsufficientFundsException(
            "VIP order requires ${order.total_amount}, have ${vip_customer.account_balance}"
        )

    # Deduct amount from balance
    vip_customer.account_balance = vip_customer.account_balance - order.total_amount

    # Update spending tracker
    vip_customer.total_spending = vip_customer.total_spending + order.total_amount

    # Add order to history
    vip_customer.order_history.append(order)

    # Log VIP order with discount
    LOG("VIPCustomer {vip_customer.user_id} placed order {order_id} with discount ${discount_amount}")

    # Return created order
    RETURN order
END FUNCTION
```

```
# ============================================================
# UC-03: Customer Order Placement
# ============================================================
# ------------------------------------------------------------
# UC-03 - Regular Customer Places Order
# Purpose: Allow registered customers to place orders with balance checking
# Input:
#     - customer: Customer object
#     - menu_items: List[MenuItem] (items to order)
#     - quantities: List[Integer] (quantity for each item)
#     - delivery_fee: Float (delivery charge, default 5.0)
# Output:
#     - order: Order object (created order)
# Exceptions:
#     - UnauthorizedAccessException: If account not active
#     - InvalidOrderException: If items/quantities mismatch
#     - InsufficientFundsException: If balance too low (also adds warning)

FUNCTION place_order_regular_customer(customer, menu_items, quantities, delivery_fee=5.0):
    # Check if customer account is active
    IF customer.account_status != ACTIVE:
        THROW UnauthorizedAccessException("Account {customer.account_status}")

    # Validate items and quantities match
    IF LENGTH(menu_items) != LENGTH(quantities):
        THROW InvalidOrderException("Items and quantities mismatch")

    # Generate unique order ID
    order_count = LENGTH(customer.order_history) + 1
    timestamp = CURRENT_TIMESTAMP()
    order_id = "ORD-{customer.user_id}-{order_count}-{timestamp}"

    # Create order items list
    order_items = []
    FOR i FROM 0 TO LENGTH(menu_items) - 1:
        item = menu_items[i]
        quantity = quantities[i]

        # Create OrderItem object
        order_item = CREATE OrderItem(
            menu_item_id=item.item_id,
            name=item.name,
            price=item.price,
            quantity=quantity
        )
        order_items.append(order_item)

    # Regular customers get no discount
    discount = 0.0

    # Create Order object
    order = CREATE Order(
        order_id=order_id,
        customer_id=customer.user_id,
        items=order_items,
        delivery_fee=delivery_fee,
        discount=discount,
        delivery_address=customer.address
    )

    # Check if customer has sufficient funds
    IF customer.account_balance < order.total_amount:
        # Add automatic warning for insufficient funds
        customer.warnings = customer.warnings + 1
        LOG("Customer {customer.user_id} received warning for insufficient funds")

        # Throw exception
        THROW InsufficientFundsException(
            "Need ${order.total_amount}, have ${customer.account_balance}"
        )

    # Deduct amount from balance
    customer.account_balance = customer.account_balance - order.total_amount

    # Update spending tracker
    customer.total_spending = customer.total_spending + order.total_amount

    # Add order to history
    customer.order_history.append(order)

    # Log successful order
    LOG("Customer {customer.user_id} placed order {order_id}")

    # Return created order
    RETURN order
END FUNCTION
```

```
# ============================================================
# UC-04: Chef & Kitchen Menu Management
# ============================================================
# ------------------------------------------------------------
# UC-04 - Chef Creates Menu Item
# Purpose: Allow chefs to create new dishes for the menu
# Input:
#     - chef: Chef object
#     - item_id: String (unique item identifier)
#     - name: String (dish name)
#     - description: String (dish description)
#     - price: Float (dish price)
#     - is_early_access: Boolean (True if VIP-only, default False)
# Output:
#     - menu_item: MenuItem object (newly created menu item)

FUNCTION create_menu_item(chef, item_id, name, description, price, is_early_access=False):
    # Create MenuItem object
    menu_item = CREATE MenuItem(
        item_id=item_id,
        name=name,
        description=description,
        price=price,
        chef_id=chef.user_id,
        is_early_access=is_early_access
    )

    # Add to chef's menu items dictionary
    chef.menu_items[item_id] = menu_item

    # Log creation
    LOG("Chef {chef.user_id} created menu item {item_id}")

    # Return created item
    RETURN menu_item
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-04 - Chef Updates Menu Item
# Purpose: Allow chefs to modify existing menu items they created
# Input:
#     - chef: Chef object
#     - item_id: String (item to update)
#     - name: String (new name, optional)
#     - description: String (new description, optional)
#     - price: Float (new price, optional)
# Output:
#     - menu_item: MenuItem object (updated item)
# Exceptions:
#     - InvalidOrderException: If menu item not found

FUNCTION update_menu_item(chef, item_id, name=None, description=None, price=None):
    # Try to get menu item from chef's items
    IF item_id NOT IN chef.menu_items:
        THROW InvalidOrderException("Menu item not found")

    # Get the menu item
    item = chef.menu_items[item_id]

    # Update fields if provided
    IF name IS NOT None:
        item.name = name

    IF description IS NOT None:
        item.description = description

    IF price IS NOT None:
        item.price = price

    # Log update
    LOG("Chef {chef.user_id} updated menu item {item_id}")

    # Return updated item
    RETURN item
END FUNCTION
```

```
    # ------------------------------------------------------------
    # UC-04 - Chef Views Feedback
    # Purpose: Allow chefs to view complaints and compliments about their dishes
    # Input:
    #     - chef: Chef object
    # Output:
    #     - feedback_list: List[Dict] (all feedback received by chef)

    FUNCTION view_chef_feedback(chef):
        # Initialize empty list
        feedback_list = []

        # Iterate through chef's feedback
        FOR each feedback IN chef.feedback_received:
            # Convert feedback to dictionary
            feedback_dict = feedback.to_dict()
            feedback_list.append(feedback_dict)

        # Return list of feedback
        RETURN feedback_list
    END FUNCTION

    # ------------------------------------------------------------
    # UC-04 - Chef Receives Order
    # Purpose: Notify chef when new order is placed
    # Input:
    #     - order: Order object (newly placed order)
    #     - chef: Chef object (chef who will prepare)
    # Output:
    #     - None (side effect: order status updated to PREPARING)

    FUNCTION receive_order_notification(order, chef):
        # Log order received in kitchen
        LOG("Kitchen received order {order.order_id} for Chef {chef.user_id}")

        # Update order status to preparing
        order.update_status(PREPARING)

        # Notify chef (in real system, this would send notification)
        DISPLAY_MESSAGE("New order {order.order_id} for preparation")
    END FUNCTION

# ============================================================
# UC-05 - Delivery Operations
# ============================================================
# ------------------------------------------------------------
# UC-05 - Assign Delivery Person to Order
# Purpose: Assign a delivery person to an order after kitchen preparation
# Input:
#     - order: Order object
#     - delivery_person_id: String (ID of delivery person)
# Output:
#     - None (side effect: order status changed to CONFIRMED, delivery person assigned)

FUNCTION assign_delivery_person(order, delivery_person_id):
    # Set delivery person ID on order
    order.delivery_person_id = delivery_person_id

    # Update order status to CONFIRMED
    order.update_status(CONFIRMED)

    # Log assignment
    LOG("Order {order.order_id} assigned to delivery person {delivery_person_id}")
END FUNCTION

# ------------------------------------------------------------
# UC-05 - Mark Order as Delivered
# Purpose: Update order status when delivery person completes delivery
# Input:
#     - order: Order object
# Output:
#     - None (side effect: order status changed to DELIVERED)

FUNCTION mark_order_delivered(order):
    # Update order status to DELIVERED
    order.update_status(DELIVERED)

    # Log delivery completion
    LOG("Order {order.order_id} marked as delivered")
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-05 - Cancel Order
# Purpose: Allow cancellation of orders that haven't been delivered yet
# Input:
#     - order: Order object
# Output:
#     - None (side effect: order status changed to CANCELLED)
# Exceptions:
#     - InvalidOrderException: If order already delivered

FUNCTION cancel_order(order):
    # Check if order already delivered
    IF order.status == DELIVERED:
        THROW InvalidOrderException("Cannot cancel a delivered order")

    # Update order status to CANCELLED
    order.update_status(CANCELLED)

    # Log cancellation
    LOG("Order {order.order_id} cancelled")
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-05 - Delivery Person Submits Bid for Order
# Purpose: Allow delivery people to bid on available orders
# Input:
#     - delivery_person: DeliveryPerson object
#     - order: Order object
#     - bid_amount: Float (delivery fee they're willing to accept)
#     - estimated_time: Integer (estimated delivery time in minutes)
# Output:
#     - bid: Bid object

FUNCTION submit_delivery_bid(delivery_person, order, bid_amount, estimated_time):
    # Create bid object
    bid = NEW Bid
    bid.bid_id = GENERATE_UNIQUE_ID("BID-")
    bid.delivery_person_id = delivery_person.person_id
    bid.order_id = order.order_id
    bid.bid_amount = bid_amount
    bid.estimated_time = estimated_time
    bid.timestamp = CURRENT_TIMESTAMP()
    bid.status = "pending"

    # Log bid submission
    LOG("Delivery person {delivery_person.person_id} bid ${bid_amount} for order {order.order_id}")

    RETURN bid
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-05 - Manager Assigns Order Based on Bids
# Purpose: Manager selects delivery person from bids (usually lowest, but can choose higher with justification)
# Input:
#     - manager: Manager object
#     - order: Order object
#     - bids: List[Bid] (all bids for this order)
#     - selected_bid_id: String (bid chosen by manager)
#     - memo: String (justification if not choosing lowest bid, optional)
# Output:
#     - assignment: OrderAssignment object
# Exceptions:
#     - ValueError: If memo required but not provided

FUNCTION assign_order_from_bids(manager, order, bids, selected_bid_id, memo=""):
    # Find the selected bid
    selected_bid = None
    FOR each bid IN bids:
        IF bid.bid_id == selected_bid_id:
            selected_bid = bid
            BREAK

    IF selected_bid IS None:
        THROW ValueError("Selected bid not found")

    # Find the lowest bid
    lowest_bid = bids[0]
    FOR each bid IN bids:
        IF bid.bid_amount < lowest_bid.bid_amount:
            lowest_bid = bid

    # Check if manager chose a higher bid
    IF selected_bid.bid_amount > lowest_bid.bid_amount:
        # Memo is REQUIRED for justification
        IF memo == "" OR memo IS None:
            THROW ValueError("Memo required when not choosing lowest bid")

        # Log justification
        LOG("Manager chose higher bid: ${selected_bid.bid_amount} vs ${lowest_bid.bid_amount}")
        LOG("Justification: {memo}")

    # Create assignment
    assignment = NEW OrderAssignment
    assignment.assignment_id = GENERATE_UNIQUE_ID("ASSIGN-")
    assignment.order_id = order.order_id
    assignment.delivery_person_id = selected_bid.delivery_person_id
    assignment.delivery_fee = selected_bid.bid_amount
    assignment.manager_id = manager.user_id
    assignment.memo = memo
    assignment.timestamp = CURRENT_TIMESTAMP()

    # Update order with delivery person
    order.delivery_person_id = selected_bid.delivery_person_id
    order.delivery_fee = selected_bid.bid_amount
    order.update_status(CONFIRMED)

    # Update bid status
    selected_bid.status = "accepted"

    # Reject other bids
    FOR each bid IN bids:
        IF bid.bid_id != selected_bid_id:
            bid.status = "rejected"

    # Log assignment
    LOG("Order {order.order_id} assigned to delivery person {selected_bid.delivery_person_id}")

    RETURN assignment
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-05 - Get All Bids for Order
# Purpose: Retrieve all bids submitted for a specific order
# Input:
#     - order_id: String (order identifier)
#     - all_bids: List[Bid] (all bids in system)
# Output:
#     - order_bids: List[Bid] (bids for this specific order)

FUNCTION get_all_bids_for_order(order_id, all_bids):
    # Initialize empty list
    order_bids = []

    # Filter bids for this order
    FOR each bid IN all_bids:
        IF bid.order_id == order_id:
            order_bids.append(bid)

    # Sort by bid amount (lowest first)
    order_bids = SORT(order_bids, BY="bid_amount", ORDER="ascending")

    # Log retrieval
    LOG("Found {LENGTH(order_bids)} bids for order {order_id}")

    RETURN order_bids
END FUNCTION

# ------------------------------------------------------------
# UC-05 - Notify Delivery Person of Assignment
# Purpose: Notify delivery person when they win a bid
# Input:
#     - delivery_person: DeliveryPerson object
#     - order: Order object
# Output:
#     - None (side effect: notification sent)

FUNCTION notify_delivery_person(delivery_person, order):
    # Create notification message
    message = "You have been assigned order {order.order_id}"
    message = message + " - Delivery fee: ${order.delivery_fee}"
    message = message + " - Address: {order.delivery_address}"

    # Send notification
    SEND_NOTIFICATION(delivery_person, message)

    # Log notification
    LOG("Notified delivery person {delivery_person.person_id} of assignment")
END FUNCTION
```

```
# ============================================================
# UC-06: Rating and Feedback System
# ============================================================
# ------------------------------------------------------------
# UC-06 - Customer Rates Dish
# Purpose: Allow customers to rate dishes, with VIP ratings weighted more heavily
# Input:
#     - customer: Customer or VIPCustomer object
#     - menu_item: MenuItem object (dish being rated)
#     - rating: Float (rating value 0-5)
#     - comment: String (optional comment)
# Output:
#     - None (side effect: menu item rating updated)
# Exceptions:
#     - InvalidRatingException: If rating not between 0 and 5

FUNCTION rate_dish(customer, menu_item, rating, comment=None):
    # Check if customer is VIP
    is_vip = ISINSTANCE(customer, VIPCustomer)

    # Validate rating range
    IF rating < 0 OR rating > 5:
        THROW InvalidRatingException("Rating must be between 0 and 5")

    # Track low and high ratings
    IF rating < 2:
        menu_item.low_rating_count = menu_item.low_rating_count + 1

    IF rating > 4:
        menu_item.high_rating_count = menu_item.high_rating_count + 1

    # Calculate weight (VIP ratings count 1.5x)
    weight = 1.5 IF is_vip ELSE 1.0

    # Calculate new weighted average
    total_weight = menu_item.total_ratings + weight
    old_weighted_sum = menu_item.rating * menu_item.total_ratings
    new_weighted_sum = old_weighted_sum + (rating * weight)
    menu_item.rating = new_weighted_sum / total_weight

    # Increment total ratings count
    menu_item.total_ratings = menu_item.total_ratings + 1

    # Log rating
    LOG("Customer {customer.user_id} rated {menu_item.item_id}: {rating}/5, VIP: {is_vip}, comment={comment}")
END FUNCTION

# ------------------------------------------------------------
# UC-06 - Customer Submits Feedback
# Purpose: Allow customers to submit complaints or compliments about dishes, delivery, or other customers
# Input:
#     - customer: Customer object
#     - feedback_type: FeedbackType (COMPLAINT or COMPLIMENT)
#     - target_type: String ("menu_item", "delivery_person", "customer")
#     - target_id: String (ID of target entity)
#     - content: String (feedback message)
# Output:
#     - feedback: Feedback object (created feedback)

FUNCTION submit_feedback(customer, feedback_type, target_type, target_id, content):
    # Check if customer is VIP
    is_vip = ISINSTANCE(customer, VIPCustomer)

    # Generate unique feedback ID
    feedback_id = GENERATE_UNIQUE_ID("FB-")

    # Create Feedback object
    feedback = CREATE Feedback(
        feedback_id=feedback_id,
        customer_id=customer.user_id,
        feedback_type=feedback_type,
        target_type=target_type,
        target_id=target_id,
        content=content,
        is_vip=is_vip
    )

    # Add to customer's submitted feedback list
    customer.feedback_submitted.append(feedback)

    # Log feedback submission
    LOG("Feedback {feedback_id} created: {feedback_type} (VIP: {is_vip})")

    # Return created feedback
    RETURN feedback
END FUNCTION
```

```
        # ------------------------------------------------------------
        # UC-06 - Customer Files Complaint on Order
        # Purpose: Mark an order as having a complaint
        # Input:
        #     - order: Order object
        # Output:
        #     - None (side effect: order.has_complaint set to True)

        FUNCTION file_complaint_on_order(order):
            # Set complaint flag on order
            order.has_complaint = True

            # Log complaint
            LOG("Complaint filed for order {order.order_id}")
        END FUNCTION


# ============================================================
# UC-07: Reputation Management
# ============================================================
# ------------------------------------------------------------
# UC-07 - Manager Reviews Feedback
# Purpose: Manager processes complaints and compliments, issuing warnings or tracking compliments
# Input:
#     - manager: Manager object
#     - feedback_list: List[Feedback] (feedback to review)
#     - customer_dict: Dict[String, Customer] (mapping of customer IDs to Customer objects)
# Output:
#     - None (side effects: warnings added, customers may be suspended, compliments tracked)

FUNCTION review_feedback_manager(manager, feedback_list, customer_dict):
    # Iterate through each feedback item
    FOR each feedback IN feedback_list:
        # Check if feedback target is a customer
        IF feedback.target_type == "customer":
            # Get customer object from dictionary
            customer = customer_dict.GET(feedback.target_id)

            IF customer IS NOT None:
                # Handle complaint
                IF feedback.feedback_type == COMPLAINT:
                    # Add warning to customer
                    customer.warnings = customer.warnings + 1
                    LOG("Manager issued warning to customer {customer.user_id}")

                    # Check if customer exceeded max warnings
                    IF customer.warnings >= MAX_WARNINGS:
                        # Suspend customer account
                        customer.account_status = SUSPENDED
                        LOG("Customer {customer.user_id} suspended due to repeated complaints")

                # Handle compliment
                ELSE IF feedback.feedback_type == COMPLIMENT:
                    # Increment compliment counter
                    customer.compliments_received = customer.compliments_received + 1
                    LOG("Customer {customer.user_id} received compliment")
END FUNCTION
```

```
    # ------------------------------------------------------------
    # UC-07 - Cancel Complaint with Compliment
    # Purpose: Allow one compliment to cancel one unresolved complaint
    # Input:
    #     - complaint_feedback: Feedback object (complaint to cancel)
    # Output:
    #     - success: Boolean (True if cancelled, False if cannot cancel)


    FUNCTION cancel_complaint_with_compliment(complaint_feedback):
        # Check all conditions for cancellation
        IF complaint_feedback.feedback_type == COMPLAINT AND
           complaint_feedback.can_cancel_complaint AND
           NOT complaint_feedback.is_resolved:

            # Mark as resolved
            complaint_feedback.is_resolved = True

            # Add cancellation note
            complaint_feedback.response = "Cancelled by compliment"

            # Log cancellation
            LOG("Complaint {complaint_feedback.feedback_id} cancelled by compliment")

            RETURN True

        # Cannot cancel
        RETURN False
    END FUNCTION

# ============================================================
# UC-08: HR & Employee Management
# ============================================================
# ------------------------------------------------------------
# UC-08 - Promote Customer to VIP
# Purpose: Promote eligible customers to VIP status ($100 spent OR 3 orders)
# Input:
#     - customer: Customer object
# Output:
#     - vip_customer: VIPCustomer object OR None if not eligible

FUNCTION promote_to_vip(customer):
    # Check eligibility criteria (OR logic: either condition works)
    IF customer.total_spending >= VIP_SPENDING_THRESHOLD OR
       LENGTH(customer.order_history) >= VIP_ORDER_THRESHOLD:

        # Create VIPCustomer from base customer
        vip = CREATE VIPCustomer(customer)

        # Copy all customer data to VIP
        vip.account_balance = customer.account_balance
        vip.total_spending = customer.total_spending
        vip.order_history = customer.order_history
        vip.warnings = customer.warnings
        vip.feedback_submitted = customer.feedback_submitted

        # Log promotion
        LOG("Customer {customer.user_id} promoted to VIP")

        RETURN vip

    # Not eligible
    RETURN None
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-08 - Demote VIP to Regular Customer
# Purpose: Demote VIP customers who receive 2 warnings back to regular customer status
# Input:
#     - vip_customer: VIPCustomer object
# Output:
#     - None (side effect: role changed to CUSTOMER, warnings cleared)

FUNCTION demote_vip_to_customer(vip_customer):
    # Check if VIP has reached warning threshold (2 warnings)
    IF vip_customer.warnings >= vip_customer.max_warnings:
        # Change role back to regular customer
        vip_customer.role = CUSTOMER

        # Clear warnings after demotion
        vip_customer.warnings = 0

        # Log demotion
        LOG("VIP {vip_customer.user_id} demoted to regular customer")
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-08 - Manager Performs HR Action
# Purpose: Manager can promote, demote, or terminate employee accounts
# Input:
#     - manager: Manager object
#     - employee: BaseUser object (Customer or Chef)
#     - action: String ("promote", "demote", "terminate")
# Output:
#     - None (side effects depend on action)
# Exceptions:
#     - UnauthorizedAccessException: If not Customer or Chef
#     - ValueError: If invalid action

FUNCTION perform_hr_action(manager, employee, action):
    # Verify employee type
    IF NOT ISINSTANCE(employee, Customer) AND NOT ISINSTANCE(employee, Chef):
        THROW UnauthorizedAccessException("HR actions only apply to Customer or Chef")

    # Handle promote action
    IF action == "promote":
        LOG("Manager promoted {employee.user_id}")

        # If employee is customer, try VIP promotion
        IF ISINSTANCE(employee, Customer):
            vip_customer = promote_to_vip(employee)
            IF vip_customer IS NOT None:
                # Replace customer with VIP in system
                # (Implementation detail: update references)
                LOG("Customer promoted to VIP")

    # Handle demote action
    ELSE IF action == "demote":
        LOG("Manager demoted {employee.user_id}")

        # If employee is VIP, demote to regular customer
        IF ISINSTANCE(employee, VIPCustomer):
            employee.role = CUSTOMER
            employee.warnings = 0

    # Handle terminate action
    ELSE IF action == "terminate":
        # Close employee account
        employee.account_status = CLOSED
        LOG("Manager terminated account for {employee.user_id}")

    # Invalid action
    ELSE:
        THROW ValueError("Invalid HR action: {action}")
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-08 - Track Chef Performance
# Purpose: Monitor chef's dish ratings and NET complaints (after cancellation)
# Input:
#     - chef: Chef object
#     - menu_items: List[MenuItem] (all chef's dishes)
# Output:
#     - performance: Dict (contains low_rating_dishes, high_rating_dishes, net_complaint_count, compliment_count)

FUNCTION track_chef_performance(chef, menu_items):
    # Initialize counters
    low_rating_dishes = 0
    high_rating_dishes = 0
    raw_complaint_count = 0
    compliment_count = 0

    # Check each dish for ratings
    FOR each item IN menu_items:
        IF item.chef_id == chef.user_id:
            # Count dishes with consistently low ratings (<2)
            IF item.rating < 2 AND item.total_ratings >= 5:
                low_rating_dishes = low_rating_dishes + 1

            # Count dishes with high ratings (>4)
            IF item.rating > 4 AND item.total_ratings >= 5:
                high_rating_dishes = high_rating_dishes + 1

    # Check feedback for complaints and compliments
    FOR each feedback IN chef.feedback_received:
        IF feedback.feedback_type == COMPLAINT AND NOT feedback.is_resolved:
            raw_complaint_count = raw_complaint_count + 1

        IF feedback.feedback_type == COMPLIMENT:
            compliment_count = compliment_count + 1

    # One compliment cancels one complaint
    net_complaint_count = MAX(0, raw_complaint_count - compliment_count)

    # Compliments used for cancellation don't count toward bonus
    bonus_eligible_compliments = MAX(0, compliment_count - raw_complaint_count)

    # Create performance summary
    performance = {
        "low_rating_dishes": low_rating_dishes,
        "high_rating_dishes": high_rating_dishes,
        "net_complaint_count": net_complaint_count,
        "bonus_eligible_compliments": bonus_eligible_compliments,
        "raw_complaint_count": raw_complaint_count,
        "total_compliment_count": compliment_count
    }

    RETURN performance
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-08 - Demote Chef due to poor performance
# Purpose: Reduce chef's salary and track demotion count
# Input:
#     - manager: Manager object
#     - chef: Chef object
#     - reason: String (reason for demotion)
# Output:
#     - None (side effects: salary reduced, demotion_count incremented)

FUNCTION demote_chef(manager, chef, reason):
    # Reduce salary by 10%
    salary_reduction = chef.salary * 0.10
    chef.salary = chef.salary - salary_reduction

    # Increment demotion counter
    chef.demotion_count = chef.demotion_count + 1

    # Log demotion
    LOG("Manager demoted Chef {chef.user_id}: {reason}. New salary: ${chef.salary}")

    # Check if chef should be fired (2 demotions)
    IF chef.demotion_count >= 2:
        fire_chef(manager, chef)
END FUNCTION


# ------------------------------------------------------------
# UC-08 - Give Chef Bonus due to excellent performance
# Purpose: Reward chef with bonus payment for high ratings or compliments
# Input:
#     - manager: Manager object
#     - chef: Chef object
#     - bonus_amount: Float (bonus payment amount)
#     - reason: String (reason for bonus)
# Output:
#     - None (side effect: bonus paid to chef)

FUNCTION give_chef_bonus(manager, chef, bonus_amount, reason):
    # Add bonus to chef's bonus tracker
    chef.total_bonuses = chef.total_bonuses + bonus_amount

    # Increment bonus count
    chef.bonus_count = chef.bonus_count + 1

    # Log bonus
    LOG("Chef {chef.user_id} received bonus of ${bonus_amount}: {reason}")

    # Send notification
    SEND_NOTIFICATION(chef, "You received a ${bonus_amount} bonus!")
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-08 - Evaluate Chef Performance
# Purpose: Automatically check chef performance and take appropriate action
# Input:
#     - manager: Manager object
#     - chef: Chef object
#     - menu_items: List[MenuItem] (all menu items)
# Output:
#     - action_taken: String (description of action taken)

FUNCTION evaluate_chef_performance(manager, chef, menu_items):
    # Get performance metrics
    performance = track_chef_performance(chef, menu_items)

    # Log performance for transparency
    LOG("Chef {chef.user_id} performance:")
    LOG("  - Low rating dishes: {performance['low_rating_dishes']}")
    LOG("  - High rating dishes: {performance['high_rating_dishes']}")
    LOG("  - Raw complaints: {performance['raw_complaint_count']}")
    LOG("  - Total compliments: {performance['total_compliment_count']}")
    LOG("  - Net complaints (after cancellation): {performance['net_complaint_count']}")

    # Check for demotion conditions (checked first)
    # Condition 1: Consistently low ratings (<2)
    IF performance["low_rating_dishes"] > 0:
        demote_chef(manager, chef, "Dishes with consistently low ratings")
        RETURN "Chef demoted due to low ratings"

    # Condition 2: 3 or more total complaints (after cancellation)
    IF performance["net_complaint_count"] >= 3:
        demote_chef(manager, chef, "Received 3 or more complaints (net)")
        RETURN "Chef demoted due to net complaints"

    # Check for bonus conditions (only if not demoted)
    actions_taken = []

    # Condition 1: High ratings (>4)
    IF performance["high_rating_dishes"] >= 1:
        bonus_amount = 500.0
        give_chef_bonus(manager, chef, bonus_amount, "Dishes with high ratings")
        actions_taken.append("Bonus for high ratings")

    # Condition 2: 3 or more bonus-eligible compliments
    IF performance["bonus_eligible_compliments"] >= 3:
        bonus_amount = 300.0
        give_chef_bonus(manager, chef, bonus_amount, "Received 3 compliments")
        actions_taken.append("Bonus for compliments")

    # Return appropriate message
    IF LENGTH(actions_taken) > 0:
        RETURN "Chef received: " + JOIN(actions_taken, " and ")
    ELSE:
        RETURN "No action taken - performance acceptable"
END FUNCTION
```

```
# ============================================================
# UC-09: Financial Management
# ============================================================
# ------------------------------------------------------------
# UC-09 - Customer Deposits Funds
# Purpose: Allow customers to add money to their account balance
# Input:
#     - customer: Customer object
#     - amount: Float (amount to deposit)
# Output:
#     - new_balance: Float (updated account balance)
# Exceptions:
#     - UnauthorizedAccessException: If account not active
#     - ValueError: If amount not positive

FUNCTION deposit_funds(customer, amount):
    # Check if account is active
    IF NOT customer.is_active():
        THROW UnauthorizedAccessException("Account {customer.account_status}")

    # Validate deposit amount
    IF amount <= 0:
        THROW ValueError("Deposit must be positive")

    # Add amount to balance
    customer.account_balance = customer.account_balance + amount

    # Log deposit
    LOG("Customer {customer.user_id} deposited ${amount}")

    # Return new balance
    RETURN customer.account_balance
END FUNCTION


    # ------------------------------------------------------------
    # UC-09 - Customer Withdraws Funds
    # Purpose: Allow customers to withdraw money from their account balance
    # Input:
    #     - customer: Customer object
    #     - amount: Float (amount to withdraw)
    # Output:
    #     - new_balance: Float (updated account balance)
    # Exceptions:
    #     - InsufficientFundsException: If balance too low

    FUNCTION withdraw_funds(customer, amount):
        # Check if sufficient balance
        IF amount > customer.account_balance:
            THROW InsufficientFundsException(
                "Insufficient funds: ${customer.account_balance}"
            )

        # Deduct amount from balance
        customer.account_balance = customer.account_balance - amount

        # Log withdrawal
        LOG("Customer {customer.user_id} withdrew ${amount}")

        # Return new balance
        RETURN customer.account_balance
    END FUNCTION
```

```
# ------------------------------------------------------------
# UC-09 - Check Account Balance
# Purpose: Allow customers to view their current account balance
# Input:
#     - customer: Customer object
# Output:
#     - balance: Float (current account balance)

FUNCTION check_balance(customer):
    # Return current balance
    RETURN customer.account_balance
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-09 - Get Spending Summary
# Purpose: Provide customer with their total spending and order count
# Input:
#     - customer: Customer object
# Output:
#     - summary: Dict (contains total_spending, order_count, average_order_value)

FUNCTION get_spending_summary(customer):
    # Calculate order count
    order_count = LENGTH(customer.order_history)

    # Calculate average order value
    IF order_count > 0:
        average_order_value = customer.total_spending / order_count
    ELSE:
        average_order_value = 0.0

    # Create summary dictionary
    summary = {
        "total_spending": customer.total_spending,
        "order_count": order_count,
        "average_order_value": average_order_value,
        "current_balance": customer.account_balance,
        "warnings": customer.warnings
    }

    # Return summary
    RETURN summary
END FUNCTION
```

```
# ============================================================
# UC-10: Account Closure
# ============================================================
# ------------------------------------------------------------
# UC-10 - Close Customer Account
# Purpose: Manager closes customer account after verifying no pending orders
# Input:
#     - manager: Manager object
#     - customer: Customer object
# Output:
#     - None (side effects: account closed, balance cleared)
# Exceptions:
#     - InvalidOrderException: If pending orders exist

FUNCTION close_customer_account(manager, customer):
    # Find all pending orders
    pending_orders = []
    FOR each order IN customer.order_history:
        IF order.status NOT IN [DELIVERED, CANCELLED]:
            pending_orders.append(order)

    # Check if any pending orders exist
    IF LENGTH(pending_orders) > 0:
        THROW InvalidOrderException(
            "Cannot close account {customer.user_id}, pending orders exist"
        )

    # Set account status to closed
    customer.account_status = CLOSED

    # Clear account balance (refund would happen separately)
    customer.account_balance = 0.0

    # Log closure
    LOG("Customer {customer.user_id} account closed")
END FUNCTION

# ------------------------------------------------------------
# UC-10 - Close Chef Account
# Purpose: Manager closes chef account and removes their menu items
# Input:
#     - manager: Manager object
#     - chef: Chef object
# Output:
#     - None (side effects: account closed, menu items cleared)

FUNCTION close_chef_account(manager, chef):
    # Clear all menu items created by chef
    chef.menu_items.clear()

    # Set account status to closed
    chef.account_status = CLOSED

    # Log closure
    LOG("Chef {chef.user_id} account closed")
END FUNCTION

  # ------------------------------------------------------------
  # UC-10 - Add Closed Customer to Blacklist
  # Purpose: Prevent kicked-out customers from registering again
  # Input:
  #     - manager: Manager object
  #     - email: String (email to blacklist)
  #     - blacklist: List[String] (current blacklist)
  # Output:
  #     - updated_blacklist: List[String] (blacklist with new email)

  FUNCTION add_to_blacklist(manager, email, blacklist):
      # Check if email already in blacklist
      IF email NOT IN blacklist:
          # Add email to blacklist
          blacklist.append(email)

          # Log blacklist addition
          LOG("Manager added {email} to blacklist")

      # Return updated blacklist
      RETURN blacklist
  END FUNCTION
```

```
# ============================================================
# UC-11: AI-Based Customer Service
# ============================================================
# ------------------------------------------------------------
# UC-11 - Query AI Customer Service
# Purpose: Answer user questions using knowledge base first, then LLM fallback
# Input:
#     - user: BaseUser object (any user type)
#     - question: String (user's question)
#     - knowledge_base: KnowledgeBase object
# Output:
#     - response: ChatResponse object (contains answer, source, needs_rating)

FUNCTION query_ai_customer_service(user, question, knowledge_base):
    # Log the query
    LOG("User {user.user_id} asked: {question}")

    # Try to find answer in local knowledge base
    kb_result = search_knowledge_base(question, knowledge_base)

    # Check if knowledge base has good answer
    IF kb_result IS NOT None AND kb_result.confidence >= 0.7:
        # Found answer in knowledge base
        LOG("Answer found in knowledge base (confidence: {kb_result.confidence})")

        # Create response from knowledge base
        response = CREATE ChatResponse(
            response_id=GENERATE_UNIQUE_ID("RESP-"),
            user_id=user.user_id,
            question=question,
            answer=kb_result.answer,
            source="knowledge_base",
            kb_entry_id=kb_result.entry_id,
            author_id=kb_result.author_id,
            confidence=kb_result.confidence,
            timestamp=CURRENT_TIMESTAMP(),
            needs_rating=True,  # KB answers require rating
            rating=None
        )

        RETURN response

    # No good answer in KB, delegate to LLM (Ollama)
    ELSE:
        LOG("No KB answer found, delegating to LLM")

        # Query Ollama LLM
        llm_answer = query_ollama_llm(question, knowledge_base)

        # Create response from LLM
        response = CREATE ChatResponse(
            response_id=GENERATE_UNIQUE_ID("RESP-"),
            user_id=user.user_id,
            question=question,
            answer=llm_answer,
            source="llm",
            kb_entry_id=None,
            author_id=None,
            confidence=None,
            timestamp=CURRENT_TIMESTAMP(),
            needs_rating=False,  # LLM answers don't need rating
            rating=None
        )

        RETURN response
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-11 - Search Local Knowledge Base
# Purpose: Search knowledge base for relevant answer to user's question
# Input:
#     - question: String (user's question)
#     - knowledge_base: KnowledgeBase object
# Output:
#     - result: KBSearchResult object OR None if no good match

FUNCTION search_knowledge_base(question, knowledge_base):
    # Convert question to lowercase for matching
    question_lower = question.LOWERCASE()

    # Initialize best match
    best_match = None
    best_score = 0.0

    # Search through all knowledge base entries
    FOR each entry IN knowledge_base.entries:
        # Skip flagged/removed entries
        IF entry.is_flagged OR entry.is_removed:
            CONTINUE

        # Calculate similarity score between question and entry
        score = calculate_similarity(question_lower, entry.question.LOWERCASE())

        # Also check keywords
        FOR each keyword IN entry.keywords:
            IF keyword IN question_lower:
                score = score + 0.1  # Boost for keyword match

        # Update best match if this is better
        IF score > best_score:
            best_score = score
            best_match = entry

    # Check if best match is good enough (confidence threshold)
    IF best_match IS NOT None AND best_score >= 0.5:
        # Create result object
        result = CREATE KBSearchResult(
            entry_id=best_match.entry_id,
            answer=best_match.answer,
            author_id=best_match.author_id,
            confidence=best_score
        )

        # Increment usage counter
        best_match.usage_count = best_match.usage_count + 1

        RETURN result

    # No good match found
    RETURN None
END FUNCTION
```

```
# ------------------------------------------------------
# UC-11 - Query Ollama LLM
# Purpose: Get answer from Ollama LLM when knowledge base has no answer
# Input:
#     - question: String (user's question)
#     - knowledge_base: KnowledgeBase object (for context)
# Output:
#     - answer: String (LLM-generated answer)

FUNCTION query_ollama_llm(question, knowledge_base):
    # Build context from knowledge base (optional)
    context = build_context_from_kb(knowledge_base)

    # Create prompt for Ollama
    prompt = "You are a helpful restaurant customer service assistant. "
    prompt = prompt + "Answer the following question about our restaurant:\n\n"
    prompt = prompt + "Question: {question}\n\n"

    IF context != "":
        prompt = prompt + "Context from our knowledge base:\n{context}\n\n"

    prompt = prompt + "Answer:"

    # Call Ollama API
    TRY:
        # Make HTTP request to Ollama
        response = MAKE_HTTP_REQUEST(
            method="POST",
            url="http://localhost:11434/api/generate",
            headers={"Content-Type": "application/json"},
            body={
                "model": "llama2",  # Or any other Ollama model
                "prompt": prompt,
                "stream": False,
                "temperature": 0.7,
                "max_tokens": 500
            }
        )

        # Parse response
        IF response.status_code == 200:
            data = PARSE_JSON(response.body)
            answer = data["response"]

            LOG("LLM generated answer for: {question}")
            RETURN answer
        ELSE:
            # Ollama error
            LOG_ERROR("Ollama API error: {response.status_code}")
            RETURN "I apologize, but I'm having trouble accessing the AI system. Please try again later or contact our staff."

    CATCH Exception as e:
        LOG_ERROR("Ollama connection error: {e}")
        RETURN "I apologize, but the AI assistant is currently unavailable. Please contact our staff for assistance."
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-11 - User Rates Knowledge Base Answer
# Purpose: Allow users to rate KB answers (0=outrageous, 1-5=quality)
# Input:
#     - user: BaseUser object
#     - response: ChatResponse object (the answer being rated)
#     - rating: Integer (0-5, where 0 means outrageous/bad)
# Output:
#     - None (side effects: rating stored, flags created if rating is 0)

FUNCTION rate_kb_answer(user, response, rating):
    # Validate rating
    IF rating < 0 OR rating > 5:
        THROW ValueError("Rating must be between 0 and 5")

    # Check if response is from knowledge base
    IF response.source != "knowledge_base":
        THROW ValueError("Can only rate knowledge base answers")

    # Check if already rated
    IF response.rating IS NOT None:
        THROW ValueError("This answer has already been rated")

    # Store rating
    response.rating = rating
    response.rated_at = CURRENT_TIMESTAMP()

    # Get the knowledge base entry
    kb_entry = GET_KB_ENTRY(response.kb_entry_id)

    # Update entry's rating statistics
    kb_entry.total_ratings = kb_entry.total_ratings + 1
    kb_entry.rating_sum = kb_entry.rating_sum + rating
    kb_entry.average_rating = kb_entry.rating_sum / kb_entry.total_ratings

    LOG("User {user.user_id} rated KB answer {response.response_id}: {rating}/5")

    # Handle outrageous rating (0 = very bad content)
    IF rating == 0:
        # Create flag for manager review
        flag = CREATE ContentFlag(
            flag_id=GENERATE_UNIQUE_ID("FLAG-"),
            kb_entry_id=kb_entry.entry_id,
            response_id=response.response_id,
            reporter_id=user.user_id,
            reason="Outrageous content (rating: 0)",
            timestamp=CURRENT_TIMESTAMP(),
            status="pending",
            manager_decision=None
        )

        # Mark KB entry as flagged
        kb_entry.is_flagged = True
        kb_entry.flag_count = kb_entry.flag_count + 1

        # Log the flag
        LOG("KB entry {kb_entry.entry_id} FLAGGED for manager review (0 rating)")

        # Notify manager
        manager = GET_MANAGER()
        SEND_NOTIFICATION(
            manager,
            "Knowledge base entry flagged: '{kb_entry.question}' received 0 rating"
        )
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-11 - Manager Reviews Flagged Content
# Purpose: Manager reviews flagged KB entries and decides to keep or remove
# Input:
#     - manager: Manager object
#     - flag: ContentFlag object (flagged content to review)
#     - decision: String ("keep" or "remove")
#     - notes: String (manager's notes about decision)
# Output:
#     - None (side effects: content removed/kept, author potentially banned)

FUNCTION manager_review_flagged_content(manager, flag, decision, notes):
    # Verify manager has permission
    IF NOT manager.has_permission("review_content"):
        THROW UnauthorizedAccessException("Manager lacks permission")

    # Get the knowledge base entry
    kb_entry = GET_KB_ENTRY(flag.kb_entry_id)

    # Update flag status
    flag.status = "reviewed"
    flag.manager_id = manager.user_id
    flag.manager_decision = decision
    flag.manager_notes = notes
    flag.reviewed_at = CURRENT_TIMESTAMP()

    LOG("Manager {manager.user_id} reviewed flag {flag.flag_id}: {decision}")

    # Handle manager decision
    IF decision == "remove":
        # Remove the KB entry
        kb_entry.is_removed = True
        kb_entry.is_flagged = False
        kb_entry.removed_at = CURRENT_TIMESTAMP()
        kb_entry.removed_by = manager.user_id

        LOG("KB entry {kb_entry.entry_id} REMOVED by manager")

        # Get the author
        author = GET_USER(kb_entry.author_id)

        # Ban author from contributing to knowledge base
        author.can_contribute_to_kb = False
        author.kb_ban_reason = "Content removed by manager: {notes}"

        LOG("User {author.user_id} BANNED from contributing to knowledge base")

        # Notify author
        SEND_NOTIFICATION(
            author,
            "Your knowledge base contribution has been removed. You are no longer able to contribute answers to the knowledge base."
        )

    ELSE IF decision == "keep":
        # Keep the entry, remove flag
        kb_entry.is_flagged = False

        LOG("KB entry {kb_entry.entry_id} kept after manager review")

        # Notify reporter
        reporter = GET_USER(flag.reporter_id)
        SEND_NOTIFICATION(
            reporter,
            "Thank you for your feedback. The manager has reviewed the content and decided to keep it."
        )

    ELSE:
        THROW ValueError("Invalid decision: {decision}")
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-11 - User Contributes to Knowledge Base
# Purpose: Allow employees and customers to add answers to knowledge base
# Input:
#     - user: BaseUser object (must not be banned)
#     - question: String (question being answered)
#     - answer: String (answer to the question)
#     - keywords: List[String] (keywords for searching)
# Output:
#     - kb_entry: KnowledgeBaseEntry object (newly created entry)
# Exceptions:
#     - UnauthorizedAccessException: If user is banned from contributing

FUNCTION contribute_to_knowledge_base(user, question, answer, keywords):
    # Check if user is allowed to contribute
    IF HASATTR(user, "can_contribute_to_kb") AND NOT user.can_contribute_to_kb:
        THROW UnauthorizedAccessException("You are not allowed to contribute to the knowledge base")

    # Create knowledge base entry
    kb_entry = CREATE KnowledgeBaseEntry(
        entry_id=GENERATE_UNIQUE_ID("KB-"),
        author_id=user.user_id,
        question=question,
        answer=answer,
        keywords=keywords,
        created_at=CURRENT_TIMESTAMP(),
        is_flagged=False,
        is_removed=False,
        usage_count=0,
        total_ratings=0,
        rating_sum=0,
        average_rating=0.0,
        flag_count=0
    )

    # Add to knowledge base
    knowledge_base.entries.append(kb_entry)

    # Log contribution
    LOG("User {user.user_id} contributed KB entry: '{question}'")

    RETURN kb_entry
END FUNCTION

# ------------------------------------------------------------
# UC-11 - Build Context from Knowledge Base
# Purpose: Extract relevant context from KB to provide to LLM
# Input:
#     - knowledge_base: KnowledgeBase object
# Output:
#     - context: String (formatted context for LLM)

FUNCTION build_context_from_kb(knowledge_base):
    # Get most popular/highly rated entries
    top_entries = []

    FOR each entry IN knowledge_base.entries:
        # Only include non-flagged, non-removed entries
        IF NOT entry.is_flagged AND NOT entry.is_removed:
            # Add if highly rated or frequently used
            IF entry.average_rating >= 4.0 OR entry.usage_count >= 10:
                top_entries.append(entry)

    # Limit to top 5 entries to avoid overwhelming LLM
    top_entries = SORT(top_entries, BY="average_rating", ORDER="descending")
    top_entries = top_entries[0:5]

    # Format context
    context = ""
    FOR each entry IN top_entries:
        context = context + "Q: {entry.question}\n"
        context = context + "A: {entry.answer}\n\n"

    RETURN context
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-11 - Calculate Text Similarity
# Purpose: Calculate similarity between two text strings for KB search
# Input:
#     - text1: String (first text)
#     - text2: String (second text)
# Output:
#     - similarity: Float (0.0 to 1.0, higher = more similar)

FUNCTION calculate_similarity(text1, text2):
    # Simple implementation: word overlap
    # In production, use cosine similarity with embeddings

    # Split into words
    words1 = SET(text1.SPLIT(" "))
    words2 = SET(text2.SPLIT(" "))

    # Remove common stop words
    stop_words = ["the", "a", "an", "is", "are", "what", "how", "where", "when"]
    words1 = words1 - stop_words
    words2 = words2 - stop_words

    # Calculate Jaccard similarity
    intersection = LENGTH(words1 INTERSECT words2)
    union = LENGTH(words1 UNION words2)

    IF union == 0:
        RETURN 0.0

    similarity = intersection / union

    RETURN similarity
END FUNCTION
```

```
# ============================================================
# UC-12: Image-Based Food Search & Recommendation
# ============================================================
# Creative Feature: Upload food image, get similar dishes from menu
# Technology: YOLO for object detection, CLIP for similarity matching
# Worth: 15% of final project grade

# ------------------------------------------------------------
# UC-12 - Search Menu by Image
# Purpose: Allow users to upload food image and find similar dishes in menu
# Input:
#     - user: BaseUser object (any user type)
#     - image_file: Image object (uploaded food photo)
#     - menu_items: List[MenuItem] (all available menu items)
# Output:
#     - results: ImageSearchResult object (contains matches and details)

FUNCTION search_menu_by_image(user, image_file, menu_items):
    # Log the image search
    LOG("User {user.user_id} performing image search")

    # Step 1: Detect food items in uploaded image using YOLO
    detected_items = detect_food_with_yolo(image_file)

    # Check if any food detected
    IF LENGTH(detected_items) == 0:
        LOG("No food detected in image")
        RETURN CREATE ImageSearchResult(
            search_id=GENERATE_UNIQUE_ID("ISEARCH-"),
            user_id=user.user_id,
            detected_items=[],
            matches=[],
            timestamp=CURRENT_TIMESTAMP(),
            success=False,
            error="No food items detected in image"
        )

    LOG("Detected {LENGTH(detected_items)} food items in image")

    # Step 2: For each detected item, find similar menu items
    all_matches = []

    FOR each detected_item IN detected_items:
        # Get image embeddings for detected item
        item_embedding = generate_image_embedding(detected_item.crop_image)

        # Compare with menu item images
        menu_matches = find_similar_menu_items(
            item_embedding,
            menu_items,
            detected_item.label,
            top_k=3  # Return top 3 matches per detected item
        )

        # Add matches with detected item info
        FOR each match IN menu_matches:
            match.detected_item_label = detected_item.label
            match.confidence = detected_item.confidence
            all_matches.append(match)

    # Step 3: Sort matches by similarity score
    all_matches = SORT(all_matches, BY="similarity_score", ORDER="descending")

    # Step 4: Remove duplicates (same menu item matched multiple times)
    unique_matches = remove_duplicate_matches(all_matches)

    # Step 5: Create search result
    result = CREATE ImageSearchResult(
        search_id=GENERATE_UNIQUE_ID("ISEARCH-"),
        user_id=user.user_id,
        detected_items=detected_items,
        matches=unique_matches,
        timestamp=CURRENT_TIMESTAMP(),
        success=True,
        error=None
    )

    # Log successful search
    LOG("Image search returned {LENGTH(unique_matches)} unique matches")

    RETURN result
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-12 - Detect Food Items with YOLO
# Purpose: Use YOLO model to detect food objects in image
# Input:
#     - image_file: Image object (uploaded photo)
# Output:
#     - detected_items: List[DetectedFoodItem] (food items found with bounding boxes)

FUNCTION detect_food_with_yolo(image_file):
    # Load YOLO model (YOLOv8 or YOLOv5)
    # Model trained on food dataset (e.g., Food-101, custom restaurant dataset)

    TRY:
        # Preprocess image
        image = LOAD_IMAGE(image_file)
        image_rgb = CONVERT_TO_RGB(image)

        # Run YOLO detection
        # Using YOLOv8 API format
        results = YOLO_MODEL.predict(
            source=image_rgb,
            conf=0.25,  # Confidence threshold
            iou=0.45,   # IoU threshold for NMS
            classes=[47, 48, 49, 50, 51, 52]  # Food classes from COCO dataset
            # Or use custom food detection model
        )

        detected_items = []

        # Process each detection
        FOR each detection IN results[0].boxes:
            # Get bounding box coordinates
            x1, y1, x2, y2 = detection.xyxy[0]

            # Get class label and confidence
            class_id = detection.cls[0]
            confidence = detection.conf[0]
            label = YOLO_MODEL.names[class_id]

            # Crop detected region
            crop_image = image[y1:y2, x1:x2]

            # Create detected item object
            detected_item = CREATE DetectedFoodItem(
                label=label,
                confidence=confidence,
                bounding_box=(x1, y1, x2, y2),
                crop_image=crop_image
            )

            detected_items.append(detected_item)

            LOG("Detected: {label} (confidence: {confidence:.2f})")

        RETURN detected_items

    CATCH Exception as e:
        LOG_ERROR("YOLO detection error: {e}")
        RETURN []
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-12 - Generate Image Embedding
# Purpose: Create vector embedding for image using CLIP model
# Input:
#     - image: Image object (cropped food image)
# Output:
#     - embedding: Vector (numerical representation of image)

FUNCTION generate_image_embedding(image):
    # Use CLIP (Contrastive Language-Image Pre-training) for embeddings
    # CLIP creates embeddings that work well for similarity matching

    TRY:
        # Preprocess image for CLIP
        preprocessed = CLIP_PREPROCESS(image)

        # Generate embedding using CLIP vision encoder
        WITH NO_GRADIENT():
            embedding = CLIP_MODEL.encode_image(preprocessed)
            # Normalize embedding
            embedding = embedding / NORM(embedding)

        # Convert to numpy array
        embedding = embedding.cpu().numpy()

        RETURN embedding

    CATCH Exception as e:
        LOG_ERROR("Embedding generation error: {e}")
        RETURN None
END FUNCTION

# ------------------------------------------------------------
# UC-12 - Find Similar Menu Items
# Purpose: Match detected food to menu items using embeddings
# Input:
#     - query_embedding: Vector (embedding of detected food)
#     - menu_items: List[MenuItem] (all menu items)
#     - detected_label: String (YOLO detected label for filtering)
#     - top_k: Integer (number of matches to return)
# Output:
#     - matches: List[MenuMatch] (similar menu items with scores)
```

```
# ------------------------------------------------------------
# UC-12 - Find Similar Menu Items
# Purpose: Match detected food to menu items using embeddings
# Input:
#     - query_embedding: Vector (embedding of detected food)
#     - menu_items: List[MenuItem] (all menu items)
#     - detected_label: String (YOLO detected label for filtering)
#     - top_k: Integer (number of matches to return)
# Output:
#     - matches: List[MenuMatch] (similar menu items with scores)

FUNCTION find_similar_menu_items(query_embedding, menu_items, detected_label, top_k=3):
    IF query_embedding IS None:
        RETURN []

    matches = []

    # Compare with each menu item
    FOR each menu_item IN menu_items:
        # Check if menu item has image embedding
        IF NOT HASATTR(menu_item, "image_embedding") OR menu_item.image_embedding IS None:
            # Generate embedding for menu item if not exists
            IF HASATTR(menu_item, "image_path") AND menu_item.image_path IS NOT None:
                menu_image = LOAD_IMAGE(menu_item.image_path)
                menu_item.image_embedding = generate_image_embedding(menu_image)
            ELSE:
                CONTINUE  # Skip items without images

        # Calculate cosine similarity
        similarity_score = COSINE_SIMILARITY(query_embedding, menu_item.image_embedding)

        # Optional: Boost score if labels match
        label_boost = 0.0
        IF detected_label.LOWERCASE() IN menu_item.name.LOWERCASE():
            label_boost = 0.1

        final_score = similarity_score + label_boost

        # Create match object
        match = CREATE MenuMatch(
            menu_item=menu_item,
            similarity_score=final_score,
            cosine_similarity=similarity_score,
            label_boost=label_boost
        )

        matches.append(match)

    # Sort by similarity score
    matches = SORT(matches, BY="similarity_score", ORDER="descending")

    # Return top K matches
    RETURN matches[0:top_k]
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-12 - Preprocess Menu Images (Batch Processing)
# Purpose: Generate embeddings for all menu items in advance
# Input:
#     - menu_items: List[MenuItem] (all menu items)
# Output:
#     - None (side effect: menu items updated with embeddings)

FUNCTION preprocess_menu_images(menu_items):
    LOG("Generating embeddings for {LENGTH(menu_items)} menu items")

    processed_count = 0

    FOR each menu_item IN menu_items:
        # Check if item has image
        IF NOT HASATTR(menu_item, "image_path") OR menu_item.image_path IS None:
            LOG("Menu item {menu_item.item_id} has no image, skipping")
            CONTINUE

        # Check if embedding already exists
        IF HASATTR(menu_item, "image_embedding") AND menu_item.image_embedding IS NOT None:
            LOG("Menu item {menu_item.item_id} already has embedding, skipping")
            CONTINUE

        TRY:
            # Load image
            image = LOAD_IMAGE(menu_item.image_path)

            # Generate embedding
            embedding = generate_image_embedding(image)

            # Store embedding
            menu_item.image_embedding = embedding

            processed_count = processed_count + 1
            LOG("Generated embedding for {menu_item.name}")

        CATCH Exception as e:
            LOG_ERROR("Failed to process {menu_item.item_id}: {e}")

    LOG("Preprocessing complete: {processed_count}/{LENGTH(menu_items)} items processed")
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-12 - Remove Duplicate Matches
# Purpose: Remove duplicate menu items from match results
# Input:
#     - matches: List[MenuMatch] (potentially duplicate matches)
# Output:
#     - unique_matches: List[MenuMatch] (deduplicated, keeping highest score)

FUNCTION remove_duplicate_matches(matches):
    # Track seen menu items
    seen_items = {}
    unique_matches = []

    FOR each match IN matches:
        item_id = match.menu_item.item_id

        # Check if already seen
        IF item_id IN seen_items:
            # Keep the one with higher score
            existing_score = seen_items[item_id].similarity_score
            IF match.similarity_score > existing_score:
                # Replace with higher scoring match
                FOR i FROM 0 TO LENGTH(unique_matches) - 1:
                    IF unique_matches[i].menu_item.item_id == item_id:
                        unique_matches[i] = match
                        seen_items[item_id] = match
                        BREAK
        ELSE:
            # First time seeing this item
            seen_items[item_id] = match
            unique_matches.append(match)

    RETURN unique_matches
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-12 - Upload Menu Item Image (Chef Function)
# Purpose: Allow chefs to upload images for their menu items
# Input:
#     - chef: Chef object
#     - menu_item_id: String (menu item to add image to)
#     - image_file: Image object (dish photo)
# Output:
#     - image_path: String (path where image was saved)
# Exceptions:
#     - InvalidOrderException: If menu item not found or not owned by chef

FUNCTION upload_menu_item_image(chef, menu_item_id, image_file):
    # Get menu item
    IF menu_item_id NOT IN chef.menu_items:
        THROW InvalidOrderException("Menu item not found or not owned by this chef")

    menu_item = chef.menu_items[menu_item_id]

    # Validate image
    IF NOT is_valid_image(image_file):
        THROW ValueError("Invalid image file")

    # Generate filename
    filename = "{menu_item_id}_{TIMESTAMP()}.jpg"
    image_path = "Backend/menu_images/{filename}"

    # Save image
    SAVE_IMAGE(image_file, image_path)

    # Update menu item
    menu_item.image_path = image_path

    # Generate embedding immediately
    image = LOAD_IMAGE(image_path)
    menu_item.image_embedding = generate_image_embedding(image)

    # Log upload
    LOG("Chef {chef.user_id} uploaded image for {menu_item_id}")

    RETURN image_path
END FUNCTION
```

```
# ------------------------------------------------------------
# UC-12 - Validate Image File
# Purpose: Check if uploaded file is valid image
# Input:
#     - image_file: File object
# Output:
#     - valid: Boolean (True if valid image)

FUNCTION is_valid_image(image_file):
    # Check file size (max 10MB)
    IF image_file.size > 10 * 1024 * 1024:
        RETURN False

    # Check file extension
    allowed_extensions = [".jpg", ".jpeg", ".png", ".webp"]
    file_extension = GET_FILE_EXTENSION(image_file.filename)

    IF file_extension NOT IN allowed_extensions:
        RETURN False

    # Try to load as image
    TRY:
        image = LOAD_IMAGE(image_file)
        RETURN True
    CATCH Exception:
        RETURN False
END FUNCTION

# ------------------------------------------------------------
# UC-12 - Calculate Cosine Similarity
# Purpose: Calculate similarity between two embedding vectors
# Input:
#     - vec1: Vector (first embedding)
#     - vec2: Vector (second embedding)
# Output:
#     - similarity: Float (0 to 1, higher = more similar)

FUNCTION cosine_similarity(vec1, vec2):
    # Compute dot product
    dot_product = SUM(vec1 * vec2)

    # Compute magnitudes
    magnitude1 = SQRT(SUM(vec1 * vec1))
    magnitude2 = SQRT(SUM(vec2 * vec2))

    # Avoid division by zero
    IF magnitude1 == 0 OR magnitude2 == 0:
        RETURN 0.0

    # Calculate cosine similarity
    similarity = dot_product / (magnitude1 * magnitude2)

    RETURN similarity
END FUNCTION
```

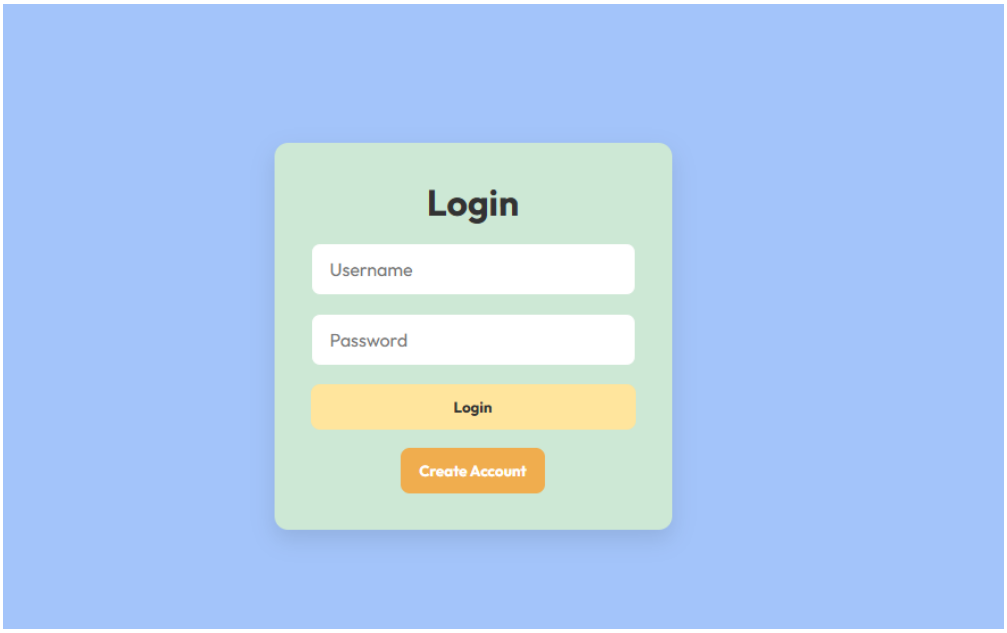
5. System Screens

5.1 GUI Screenshots

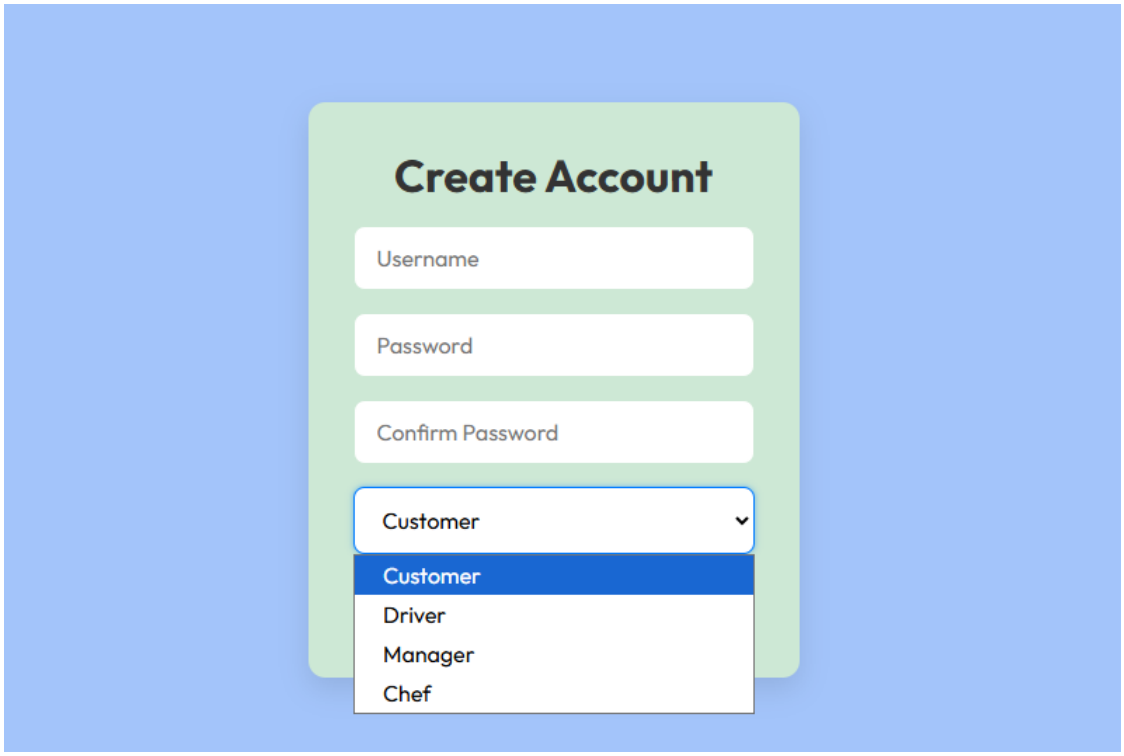
DishIQ's GUI is displayed below with labeled captions along with our logo and color scheme.
Note: The Delivery Dashboard is a work in progress right now as well as other Manager functionalities!

This is the logo for our platform DishIQ

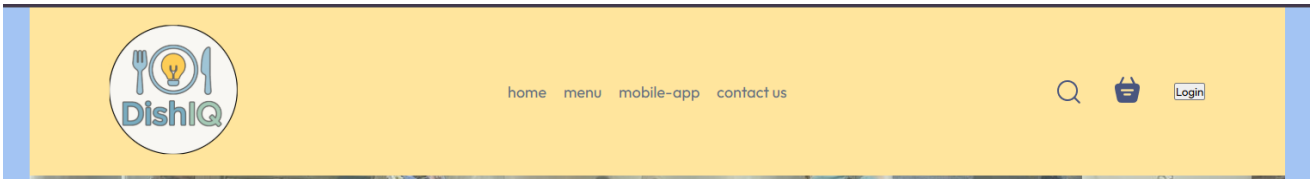
This is the color scheme used for DishIQ's GUI



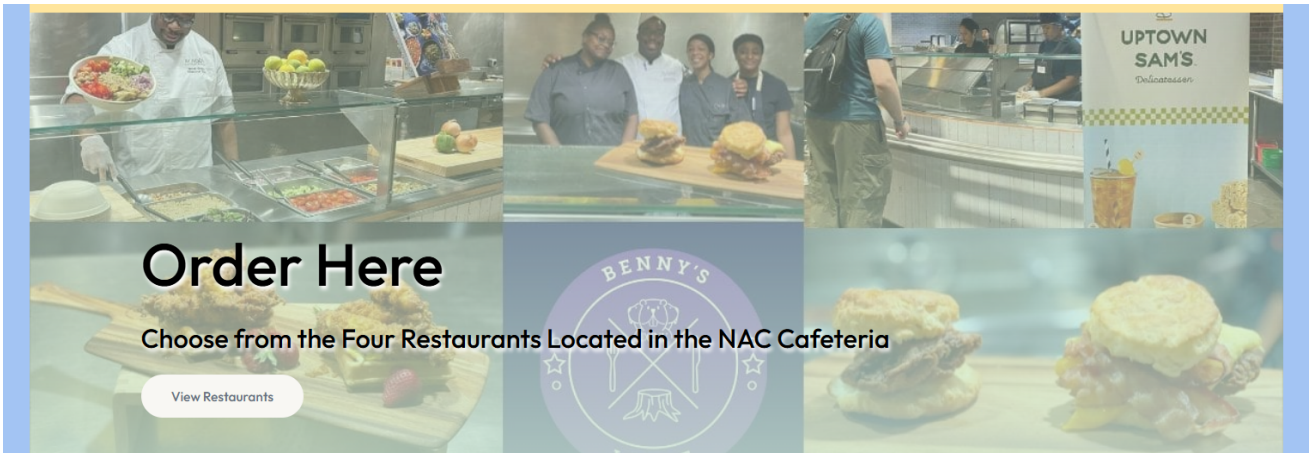
Upon clicking the Login Button present on the home screen, users are taken to this screen to login.



If users don’t already have an account, they can press the “Create Account" button, and they will be brought to this page. They create a username, password, and choose the type of account that they need.

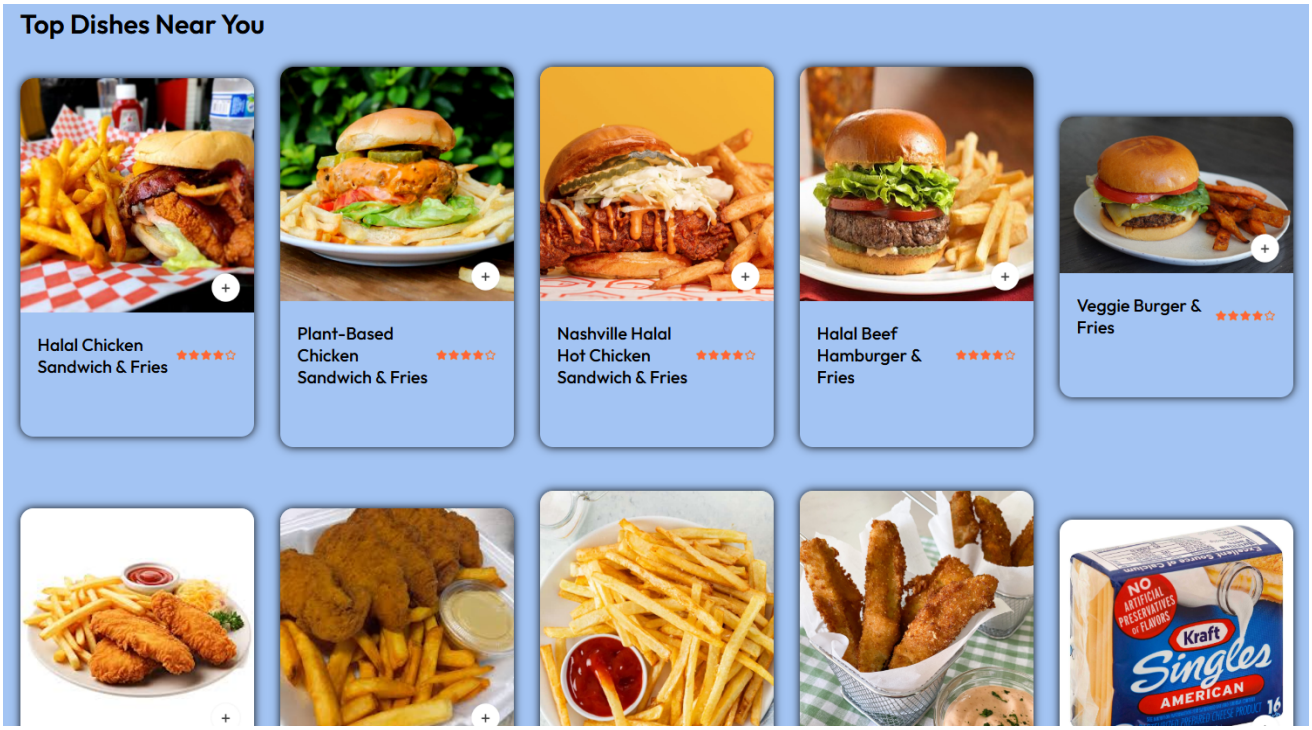


This is the navigation bar that the customer will see while using DishIQ. It helps them navigate through the platform.

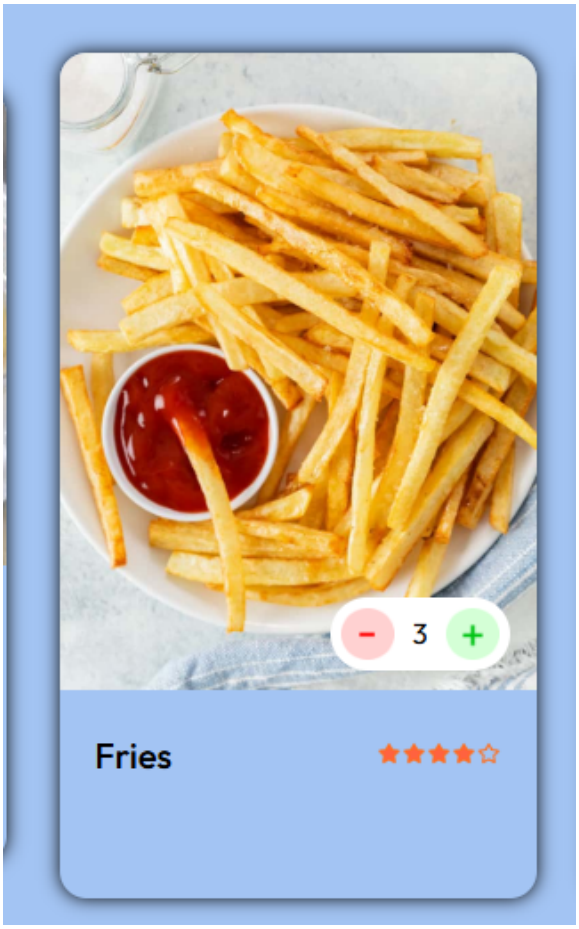


This is the header of the website.

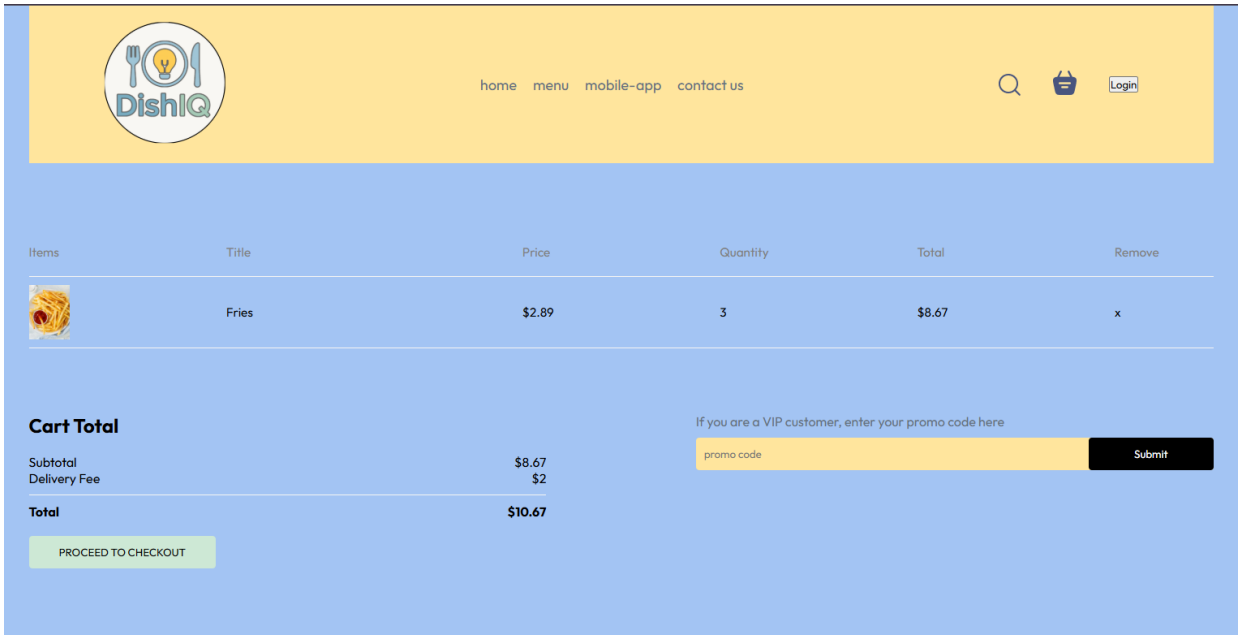
These are the restaurants and/or food options that restaurants can order from. Upon clicking on the restaurant, they can view the items on the menu.



This is the menu for Citizen Chicken. The plus button allows users to add items to their cart.

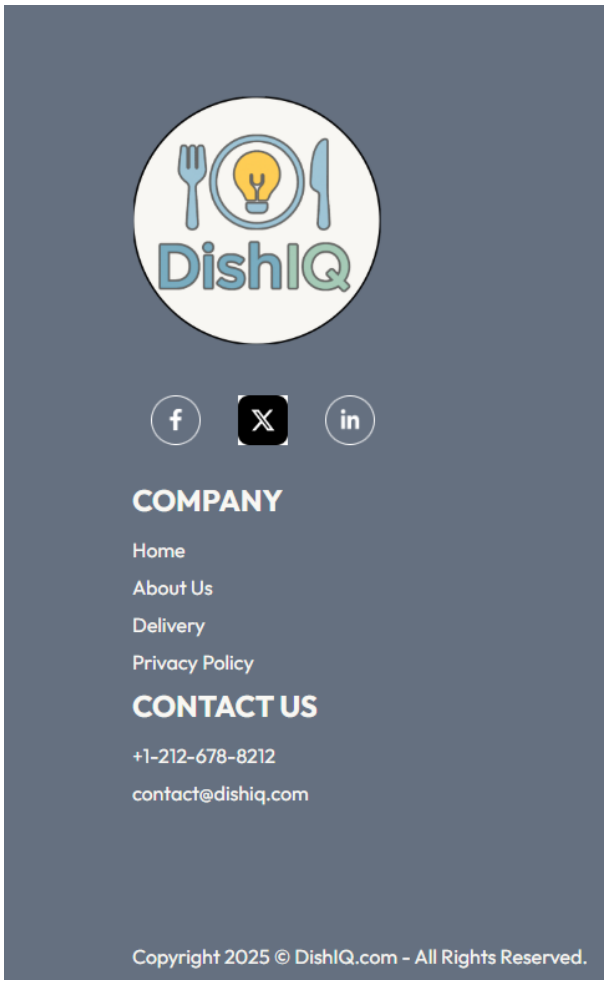


Users can press the red minus button to lessen the number of items they receive and the green plus button to increase the number of the menu item that they want.

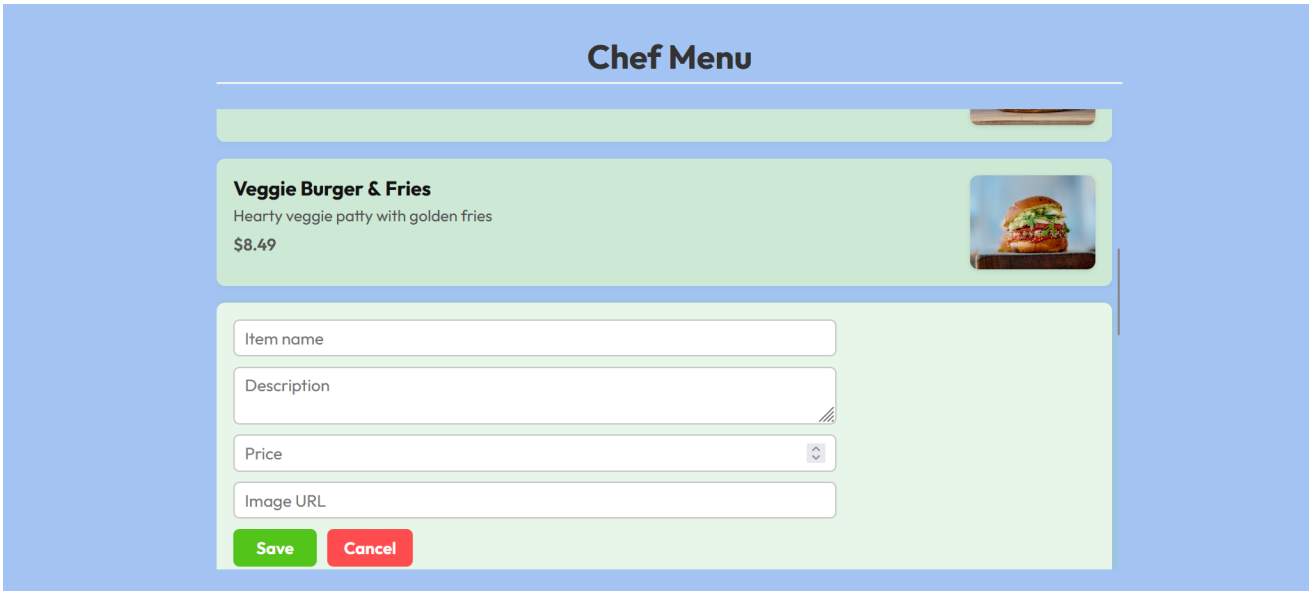


After selecting the menu items that they want, the user can press the basket icon located at the top right corner of the navigation bar. This will take them to this screen where they can see information about what items they have selected. VIP customers will have access to promo codes that will allow them to have access to deals offered by the restaurant.

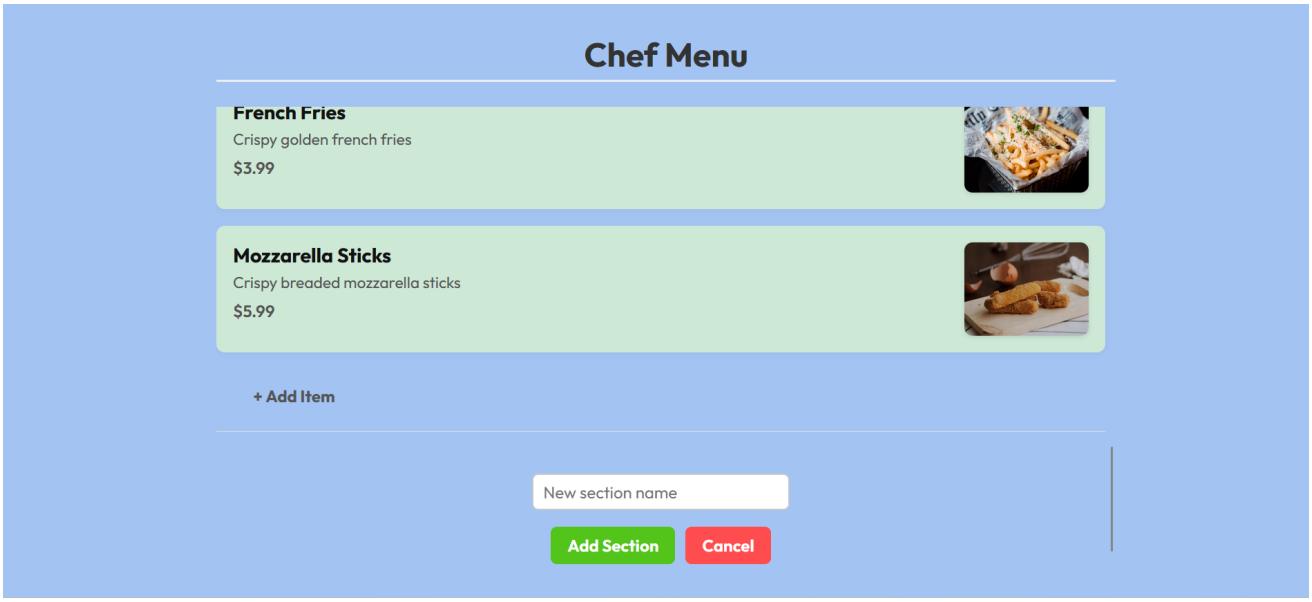
After clicking the “Proceed to Checkout” button on the cart page, users will be taken to this page. Users will input relevant information that will allow for their food to be delivered to them.



This is the footer of the website.



Users with manager/chef roles can add items to the menu



Chef and Managers roles can also add new sections

Chef and managers roles can also view and respond to reviews

They can also see the ratings of chefs

6. Memos of Group Meetings

Meeting #1
Location: Grove School of Engineering Lounge
Date: 10/28/2025
Duration: 1 hour

Meeting Summary:
In this meeting, we discussed the overall scope of Dish IQ and its functionalities to ensure comprehension from all members. Then each member shared their respective technical expertise. We allocated the division on labor based on each member's skills. Krista Singh and Lin Finnegan will work on the backend of the application using Python. Brianna Hayles and Jing Han Lin are responsible for the frontend using React JS. Marina Morcos will design and implement the database using PostgreSQL and hosting it on Supabase.

Meeting #2
Location: Grove School of Engineering Lounge
Date: 11/04/2025
Duration: 1 hour and 30 minutes

Meeting Summary:
The first objective we addressed in our meeting was to establish channels of communication and set recurring meeting times. For our main form of communication we will use the platform Discord, for its versatility in communication through video calls and messages while being accessible across many devices. As for our established meetings, we plan to meet every Tuesday and Thursday from 3:30 - 5pm at the Grove School of Engineering Lounge. A challenge we anticipate encountering is keeping track of progress and remaining at the same pace. To counteract this, Krista Singh created a notion document that contains information relating to Dish IQ's tech stack, feature details, team roles, and development internal deadlines. By maintaining this document, we can track our progress and be on the same page. The final action items of this meeting were to set goals with their respective internal timelines to ensure consistent progress. Lin Finnegan was tasked with developing the pseudocode of the backend system. Krista Singh was responsible for researching how to implement the creative feature and making the collaboration and use case diagrams. Brianna Hayles and Jing Han Lin had to develop a prototype of the frontend of Dish IQ. Marina Morcos was tasked with developing the complete database on Supabase and making the ER diagram. We aimed to finish our action items by November 15, 2025, to allow for revisions and adjustments to be made.

7. GitHub Repository

The following link is DishIQ's repository: https://github.com/KristaUSingh/DishIQ

Note: As of right now, no files are placed in the main branch, except the two reports in a folder called "Software Specification Requirements". All the code for DishIQ thus far is done in individual branches based on each person's role. All code will be merged into main once platform is completed!