Cover sheet for submission of work for assessment

UNIT DETAILS

Unit name	Software Architecture and Design			Class day/time	Sat 1:00PM
Unit code	SWE30003	Assignment no.	2	Due date	14/7/24 11:59PM
Name of lecturer/teacher	Dr. Trung Luu				
Tutor/marker's name	Dr. Trung Luu				

Office use only

Faculty or school date stamp

STUDENT(S)

	Family Name(s)	Given Name(s)	Student ID Number(s)
(1)	Truong	Duc Sang	104220420
(2)	Trinh	Quy Khang	104212003
(3)	Nguyen	Trung Kien	104169439
(4)	Le	Gia Hung	
(5)			
(6)			

DECLARATION AND STATEMENT OF AUTHORSHIP

1. I/we have not impersonated, or allowed myself/ourselves to be impersonated by any person for the purposes of this assessment.
2. This assessment is my/our original work and no part of it has been copied from any other source except where due acknowledgement is made.
3. No part of this assessment has been written for me/us by any other person except where such collaboration has been authorised by the lecturer/teacher concerned.
4. I/we have not previously submitted this work for this or any other course/unit.
5. I/we give permission for my/our assessment response to be reproduced, communicated, compared and archived for plagiarism detection, benchmarking or educational purposes.

I/we understand that:

6. Plagiarism is the presentation of the work, idea or creation of another person as though it is your own. It is a form of cheating and is a very serious academic offence that may lead to exclusion from the University. Plagiarised material can be drawn from, and presented in, written, graphic and visual form, including electronic data and oral presentations. Plagiarism occurs when the origin of the material used is not appropriately cited.

Student signature/s

I/we declare that I/we have read and understood the declaration and statement of authorship.

(1)	Sang	(4)	
(2)	Khang	(5)	
(3)	Kien	(6)	

Further information relating to the penalties for plagiarism, which range from a formal caution to expulsion from the University is contained on the Current Students website at **www.swin.edu.au/student/**

Copies of this form can be downloaded from the Student Forms web page at **www.swinburne.edu.au/studentforms/**

Executive Summary

Relaxing Koala, a café/restaurant, has been managing its day-to-day operations using low-tech, mostly manual methods. This includes taking orders from guests, passing them to the kitchen, and handling accounting. These manual processes have raised a problem that they are no longer scalable with the café/restaurant's new capacity. To address this, there is a need to introduce an information system to support daily operations.

The information system introduced is intended to streamline several key processes: managing reservations, taking customer orders, communicating orders to the kitchen, generating invoices and receipts, and handling payments.

This report includes a thorough analysis of the current operational challenges and outlines a system design for the proposed information system. Based on the previous requirement specification, some features mentioned in the case study will not be addressed, which will be detailed later. The solution features a fully automated system with customizable settings, allowing the restaurant to adapt the system to its specific needs. The report presents candidate class specifications, discusses design quality and patterns, provides an overview of the implementation process, and includes design verification.

Contents

I. Introduction

This object design document focuses on verifying the software architecture of the system. It includes a thorough problem analysis, defining classes and interfaces, and detailing the necessary outlines for the system. Serving as a reference for developers, managers, and stakeholders, this document ensures that the high-level structure of the software meets specified requirements and standards. By providing a clear big picture of how the software is intended to work, it facilitates information exchange and allows for thorough reviews and inspections to confirm that the design is correctly implemented.

1.2. Outlook of the solution

The proposed solution is a back-end REST API operating within a three-layer architecture (Presentation Layer – Front end/Client, Application Layer – Back end/API, Data Layer – Database). This API will manage the business logic atop a MySQL database, which will be limited to executing predefined table structures and merge table operations as defined by the business logic.

The software will employ serverless architecture. Upon invocation, it will establish a pool of database connections, adjustable based on business requirements. These connections will facilitate data access, after which the database will transfer data to the relevant objects. These objects will process the data and generate user responses in JSON format.

1.3. Boundary Cases

Beside functional requirements, there are still some boundaries for ine input of the back end server to function in a real life scenerio, which defined as follows:

a. Numeric boundaries

- Minimum input of a price should be 0.1AUD, a dish's preparation time should be 2 minutes.
- Maximum value of a price should be 100AUD, while a dish's preparation time should be under 40 minutes.

b. Input length

- Defined by the database, all name fields should not exceed 255 characters.
- Some special cases, such as password, username, etc should stay in a limit of 20 characters on username and up to 30 characters for passwords.

c. Resource limit

- As described, the system is expected to store illustrations images, however, those images should not exceed 10mb per image.

d. Time input limit

- Limit of the time are defined in order time, reservation time, and select report by time. Detailed, order time cannot be too far in the future or in the past, reservation time must stay 1 week in the future and cannot be in the past, start date of report time cannot be in the future, and end date of report cannot be earlier than start date, and the furthest must be in the same day.

1.4. Invalid Login Credentials Errors

To enhance the application's security, the following invalid scenarios must be addressed when implementing the login feature:

- **No token found**: This occurs when there is no authentication token provided in the secured API endpoint.
- **Invalid token**: This occurs when the authentication token used in the secured API endpoint is either fake or expired.
- **User Role Invalid**: This occurs when the user's account lacks the authorization required for the operation.
- **Invalid user**: This occurs when the token is valid, but the user no longer exists in the database. This prevents former employees from gaining unauthorized access to the server.

- **Invalid account or password**: This occurs when the account is not found or the password is incorrectly entered.

1.5. Invalid Data Case

The invalid case will be triggered if:

- **Invalid data type**: Occurs when an incorrect data type is entered, such as an invalid date, or a character entered for price or dish preparation time.
- **Missing crucial fields**: Occurs when essential fields are missing, such as a dish missing its name or illustration image.
- **Invalid linking data**: Occurs when there are invalid references, such as an order linked to a deleted dish or a user assigned to an undefined branch or location of the restaurant.

1.6. Naming conventions

Identifier	Rule	Example
Class	Capitalised camel case naming, with simple name, usually describe the database entity it interracts with	Class Dish; Class Menu;
Function / Method	Capitalised camel case, with descriptive names.	Function GetLocationByID; Function GetDishesByMenu;
Variable	Short and descriptive, match its field in the database (If exist).	Const userID; Const name;

II. Problem analysis

The Software Requirements Specification (SRS) outlines a structured approach for software development by defining a set of requirements and assigning different priority levels to each. It includes both functional and quality requirements. The document identifies key functionalities necessary to meet basic use cases, with additional features to address any gaps:

User Roles and System Focus

The system categorizes users into four roles: Waiter, Chef, Owner, and Customer. The current version focuses on assisting daily operations, which involve the waiter in taking orders, publicly displaying restaurant information, and managing restaurant operations. The owner also serves as the manager.

Management Functions

The system includes the following management functions:

- **Receipt Management**: Waiter
- **Ordered Dish List**: Waiter, Chef
- **Reservation Management**: Waiter
- **User (Employee) Management**: Owner
- **Order Management**: Waiter
- **Category (Dish) Management**: Waiter
- **Table Management**: Waiter
- **Location (Branch) Management**: Owner
- **Dish Management (Availability, Adding/Removing)**: Waiter
- **Menu Management**: Owner

Information Display Functions (No Role Required)

- Menu
- Dishes
- Category
- Location (Branch)

Removed Functionality

Certain functionalities have been removed from the scope for the following reasons:

- **Inventory Management (including tasks and materials)**: Not required by the SRS. It was initially considered because it is a common feature in restaurant management systems, but it appears to be redundant.
- **Online Account, Delivery, Reviews, Loyalty Points, and Online Ordering**: These were low-priority as the stakeholders mentioned the online function as "possibly arrange". Additionally, the SRS assumes it would be challenging for the restaurant to adopt an extensive information system, so these redundant components are removed to enhance simplicity and adoptability.

2.1. Assumptions

- **A1:** Orders are placed by waiters only; online orders are not supported in this version.
- **A2:** A unique identifier will be generated for all entities in the database layer.
- **A3:** A category must have a name and description upon creation.
- **A4:** A dining table must have a capacity, location (within the branch), and locationID (referencing the branch the table is in) upon creation.
- **A5:** A dish must have a name, description, price, preparation time, and categoryID upon creation.
- **A6:** A location must have an address, city, zip code, state, and country upon creation.
- **A7:** A menu is associated with a location (branch), as dishes served may vary between branches.
- **A8:** A dish's status on the menu (available/unavailable) can be updated by the chef if there is a kitchen issue (e.g., out of ingredients).
- **A9:** A menu must have a name, description, and locationID upon creation.
- **A10:** An order must have a tableID, status (pending, preparing, served, completed), and locationID (branch where the order is placed) upon creation.
- **A11:** A user must have a name, role, login, password, and locationID (branch where the user works) upon creation.
- **A12:** A receipt must be associated with an order and include the payment method, payment time, and locationID (branch where the order is placed) upon creation.
- **A13:** A reservation must be associated with a table and include the reservation time, status, and location upon creation.
- **A14:** Branch-related resources (location, receipt, menu, reservation, user, order, table) must be isolated. Orders from one branch cannot reach another.
- **A15:** Reservations can only be made up to one week in advance.
- **A16:** The maximum price for a dish is 100 AUD, and the minimum is 0.1 AUD.
- **A17:** The maximum preparation time for a dish is 40 minutes, and the minimum is 2 minutes.
- **A18:** A receipt is attached to only one order.
- **A19:** Reservations are made by waiters only; customers must contact a waiter to reserve a place in this version.
- **A20:** System access frequency may vary depending on the day (weekdays, weekends) and time (working hours, break hours, meal hours, etc.).

- **A21:** User passwords, once set, cannot be read as they are encrypted and cannot be decrypted due to hash technology.
- **A22:** Chefs do not need to know which order a dish belongs to; they only need a list of dishes to prepare.
- **A23:** All dates and times updated will automatically be set to the current timezone of the database.
- **A24:** Each branch can customize its setup (number of tables, menu, available dishes).
- **A25:** Delivery is not supported in this version.
- **A26:** Inventory management is not supported in this version.
- **A27:** Loyalty and discount programs are not supported in this version.
- **A28:** Reports on sales, products, etc., can only be viewed by the owner.
- **A29:** Each branch operates separately without centralized management (menu, dishes, orders).
- **A30:** The system is not integrated with any payment gateway; payments are executed manually.
- **A31:** Referenced data from other class (Table reference location, dish reference category) must be available for access before being referenced.

2.2. Simplification

Based on the initial assumptions, the project can be simplified as follows:

- **Location (Branch) Management**: Each branch is directly associated with the entity, eliminating the need to create a new table whenever a new branch is introduced.
- **Decentralized Branch Management**: Branch management will be decentralized to simplify the overall system architecture.
- **Unified User Accounts**: All user accounts are stored in the same database, differentiated by a "roles" column. This column verifies the user's role, reducing the need to rewrite similar functions for different roles.

2.3. Design Justification

A MySQL database is used for data storage in the design, and all database related operation will be limit to CRUD only, as JOIN operators are replaced by creting Views in the database.

The design follows a serverless architecture, enhancing scalability and cost efficiency, as the stakeholders are only charged on demand of their usage. Furthermore, the software follows the MVC model (Model – View – Controller) with an additional layer Middleware to handle authenication of back end and front end, make the software modular and easier to manage and scale.

2.4. Discarded Class List

- **Delivery**: Managing a logistics system is too complex for the initial software version and does not provide significant business value.
- **Points**: Customer loyalty points are not included as the current version focuses solely on assisting restaurant operations.
- **Materials**: Inventory management is complex and may confuse stakeholders. It is not specifically required by the stakeholders and, though common in management systems, is not essential in the current system.
- **Payment**: Payments are handled manually in this version to avoid unnecessary complexity, considering that end-users may not be familiar with technology.
- **Customer**: Creating customer accounts is redundant as the current version aims to assist with daily restaurant operations, not customer management.

III. Candidate Class

3.1. Candidate Class List

- Location
- Report
- DiningTable
- Reservation
- Dish
- Category
- Menu
- Order
- Receipt
- Account
    - Owner
    - Chef
    - Waiter

3.2. UML Diagram

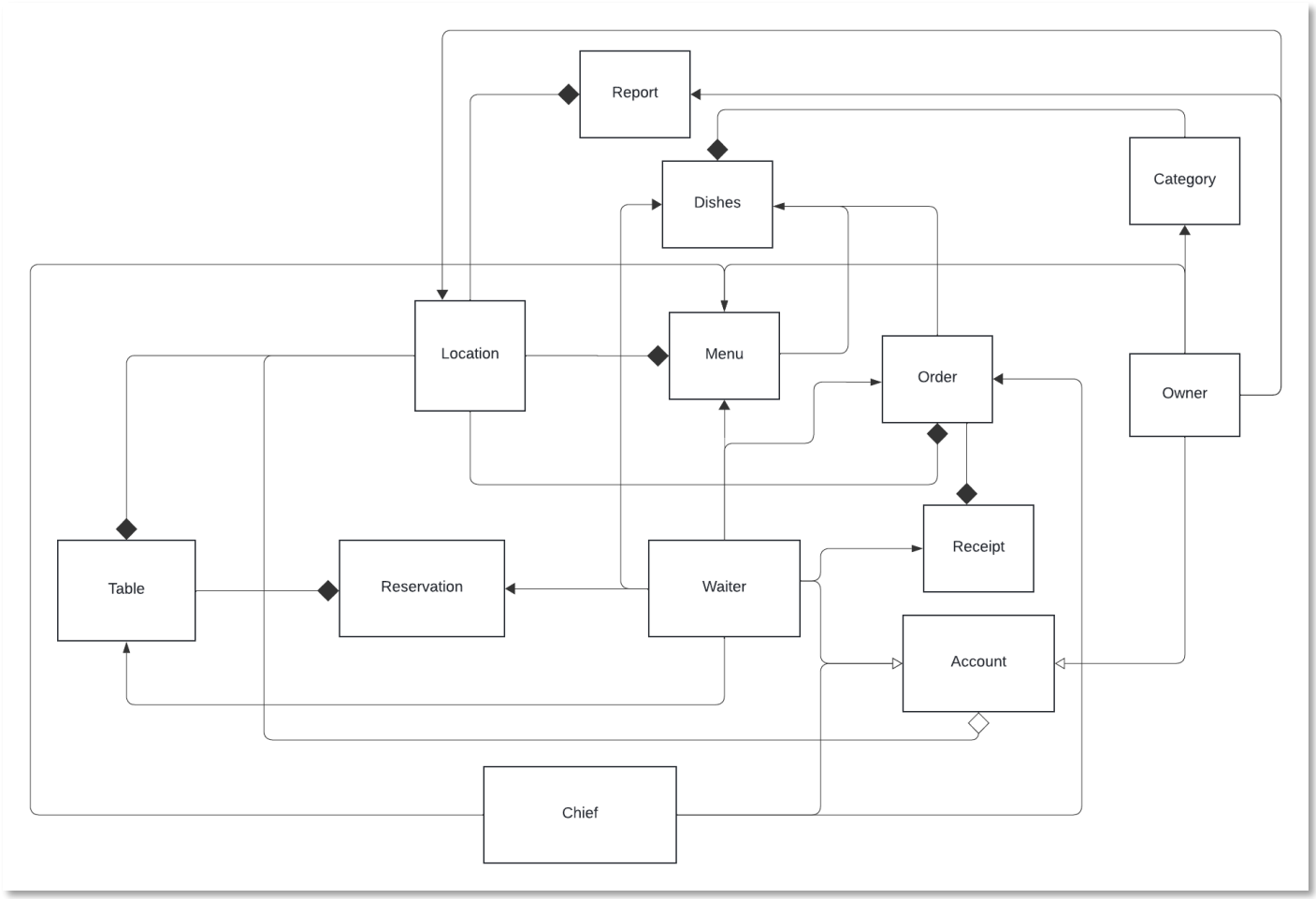

Figure 1: Class UML Diagram

The system is designed in a decentralized manner, where each branch manages itself. Consequently, certain entities cannot be shared, such as dining tables, menus, reports, and orders, which must all be associated with a specific location. However, dish data is shared among branches for easier management. Each dish must be associated with a category to facilitate menu management and structured display. Receipts must be linked to an order.

Other class relationships include:

- A menu should have a waiter responsible for managing it, including adding dishes.
- An account should be associated with a location, though this is not necessary for owner accounts.
- A menu may contain dishes or be empty, as dishes exist independently and are shared across branches. Only dishes added to the local menu are available for service.

The fundamental class in the design is the location class, which isolates resources among branches to avoid conflicts, such as orders from one branch being processed by another branch’s kitchen.

3.3. CRC Cards

3.3.1. Account

Class name: Account	
Description: *Represents a user in the system. Currently, it supports employee accounts but is designed to scale to include customer users.*	
Responsibility	**Collaborator**
User can login, and change their credentials	none
User can view and update their profile	none
Show the account's associated locationID	Location

3.3.2. Chef

Class name: Chef	
Description: *Represents a kitchen staff member who works in the kitchen area to prepare dishes.*	
Responsibility	**Collaborator**
Chef mark a dish in the order is completed	Order
Chef mark a dish in the menu to unavailable (due to out of ingredients)	Menu

3.3.3. Owner

Class name: Owner	
Description: *Represents an owner account that manages all aspects of the restaurant. An owner can also view the system under other roles to monitor work and provide feedback for system updates*.	
Responsibility	**Collaborator**
Login as Chef / Waiter	Chef, Waiter
Manage other people's account, create and delete	Account, Location (for update and create account associated with one location)
Create and manage a new dish category	Category
Create and manage a new location (branch)	Location
Create and manage a new menu	Menu
Create and manage revenue reports	Report

3.3.4 Waiter

Class name: Waiter	
Description: *Represents a restaurant staff member responsible for running the daily operations of the restaurant.*	
Responsibility	**Collaborator**
Manage table in a location	Table
Manage dishes in the shared database	Dish
Manage menu (including add dishes to menu, and excluding create or delete menu)	Menu
Manage orders	Order
Manage receipts	Receipt
Manage reservation	Reservation

3.3.5. Reservation

Class name: Reservation	
Description: *Represent a reservation made for a table in a location.*	
Responsibility	**Collaborator**
Show all reservation made (sometimes in a location)	none
Find a reservation in a table	none
Associate with a table in a time frame	Table

3.3.6. Dining Table

Class name: Dining Table	
Description: *Represent a table in a location*	
Responsibility	**Collaborator**
Display the tables within a location	Location

3.3.7. Category

Class name: Category	
Description: *Represent dish's category (i.e. Dessert, drinks, etc)*	
Responsibility	**Collaborator**
Display all available category	none

3.3.8. Location

Class name: Location	
Description: *Represent a restaurant's location*	
Responsibility	**Collaborator**
Display all available location	none

3.3.9. Menu

Class name: Menu	
Description: *Represent a menu at a specific location*	
Responsibility	**Collaborator**
Display all available menu within the location	Location
Display all dish in the menu	Dish

3.3.10. Order

Class name: Order	
Description: *Represent a customer's order*	
Responsibility	**Collaborator**
Display all orders within the location	Location
Display all dishes currently ordered (for chef)	Dish
Display order by ID	none
Display all dishes in an order	Dish

3.3.11. Dishes

Class name: Dishes	
Description: *Represents a dish in the restaurant. A dish may not be on the menu of one location but can appear on another's menu.*	
Responsibility	**Collaborator**
Display dishes	none
Display dishes in a category	Category

3.3.12. Receipt

Class name: Receipt	
Description: *Represents a receipt at the end of an order.*	
Responsibility	**Collaborator**
Display all order item and summarize the price	Order

3.3.13. Report

Class name: Report	
Description: *Represents a saved report within a given time frame.*	
Responsibility	**Collaborator**
Display summary information in a given time frame in a restaurant's location	Location

IV. Design Quality

4.1. Design Heuristics

- **H1:** A class should have a single responsibility, meaning it can only perform logical operations within its defined scope.
- **H2:** Error handling should return simple error messages to enhance security, while detailed logs should be kept safely on the server side.
- **H3:** Design should follow a serverless architecture, running only on demand to avoid unnecessary costs.
- **H4:** Utilize a pool of connections within functions to reduce bottlenecks from limited database connections.
- **H5:** Enhance security by authorizing using JWT.
- **H6:** Validate inputs to prevent injection attacks.
- **H7:** Basic cyber attacks should be prevented using provided libraries and common methods (reduce fingerprint and use helmet).
- **H8:** Proper methods should be used for each endpoint to fit their operations (PUT, PATCH, DELETE, GET).
- **H9:** Use environment variables to enable easy redeployment and enhance security.
- **H10:** Endpoint responses should be documented using Swagger.

V. Design Patterns

5.1. MVC (Model – View – Controller) Pattern

The MVC pattern separates an application into three interconnected components: Model, View, and Controller. The Model represents the data and business logic of the application. The View is responsible for displaying the data to the user, typically in JSON format in our case. The Controller handles user input and updates the Model accordingly. This separation helps in managing complex applications by organizing the code into distinct, manageable sections.

5.2. Intercepting Filter Pattern

This pattern involves processing client requests through a series of filters before reaching the main application logic. Each filter performs a specific check, such as verifying if the client is authorized, validating the authorization token, confirming the user is still in the database, and checking if the user has permission to perform the requested action. This ensures that only valid and authorized requests are processed further.

5.3. Singleton Pattern

The Singleton pattern ensures that a class has only one instance throughout the program's execution. This is particularly useful for managing resources like database connections. In our system, the database connection pool is implemented as a Singleton, meaning only one instance is created and used across the entire application to manage database connections efficiently.

5.4. Factory Pattern

The Factory pattern is used to create objects without specifying the exact class of the object that will be created. In our system, this pattern is used to generate different user roles (e.g., Chef, Waiter, Owner) from a base account class. This approach enhances code maintainability and modifiability, allowing for the easy addition of new roles like Manager and Customer as the system evolves.

VI. Bootstrap process

The following bootstrap the process of a waiter creating an order and adding a dish to the order:

1. Account class checks for token existence, returns an error if the token is not found.
2. Account class verifies the token, returns an error if the token is invalid.
3. Account class verifies the user’s ID in the token against the existing user database, returns an error if the user is not found.
4. Account class forwards the request to the Waiter class.

5. Waiter class checks for role validity, returns an error if the role is invalid.
6. Waiter class passes the request to the Order class.
7. Order class is initialized with its information.
8. Order class checks its own information, specifically verifying if the associated order location exists in the database, and returns an error if the location does not exist.
9. Order class creates a dish based on the request and verifies the dish's existence in the database and its availability at the current location. If the dish does not exist in the database or is not available at the current location, returns an error.
10. Order class adds the dish to its dish list and stores it in the database.

VII. Verification (Sequence diagram)

The process of design has been verified by the following simulated use cases:

7.1. Create a new dish in system

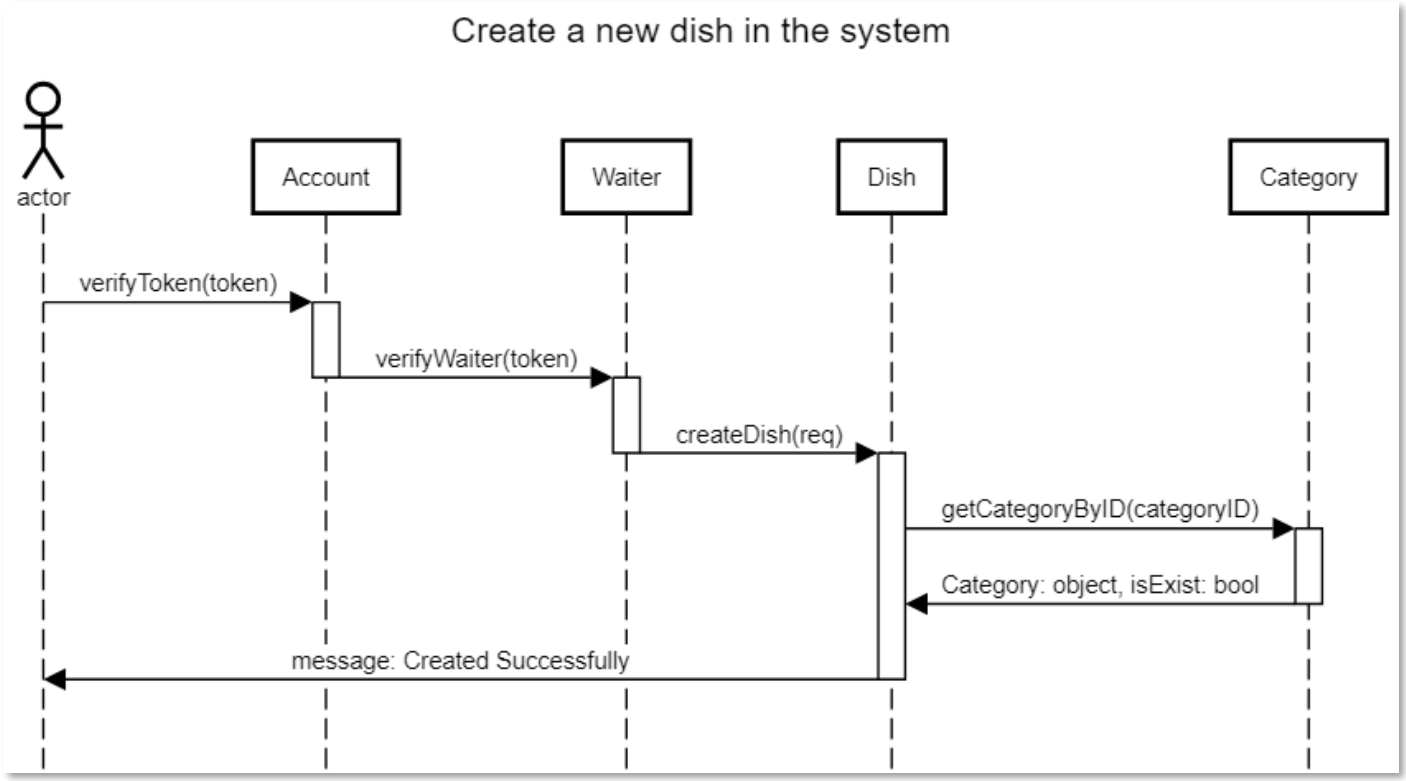

Figure 2: Sequence diagram on new dish creation in the system

To add a new dish to the system, the user first accesses the management website, which verifies their token and role. The create dish request is then passed to the dish object, which ensures the attached category is available in the database by checking its existence. If the category exists, the dish object continues executing the create dish method. Finally, a success message is returned to the user.

7.2. Owner assign account (Employee) to a new location

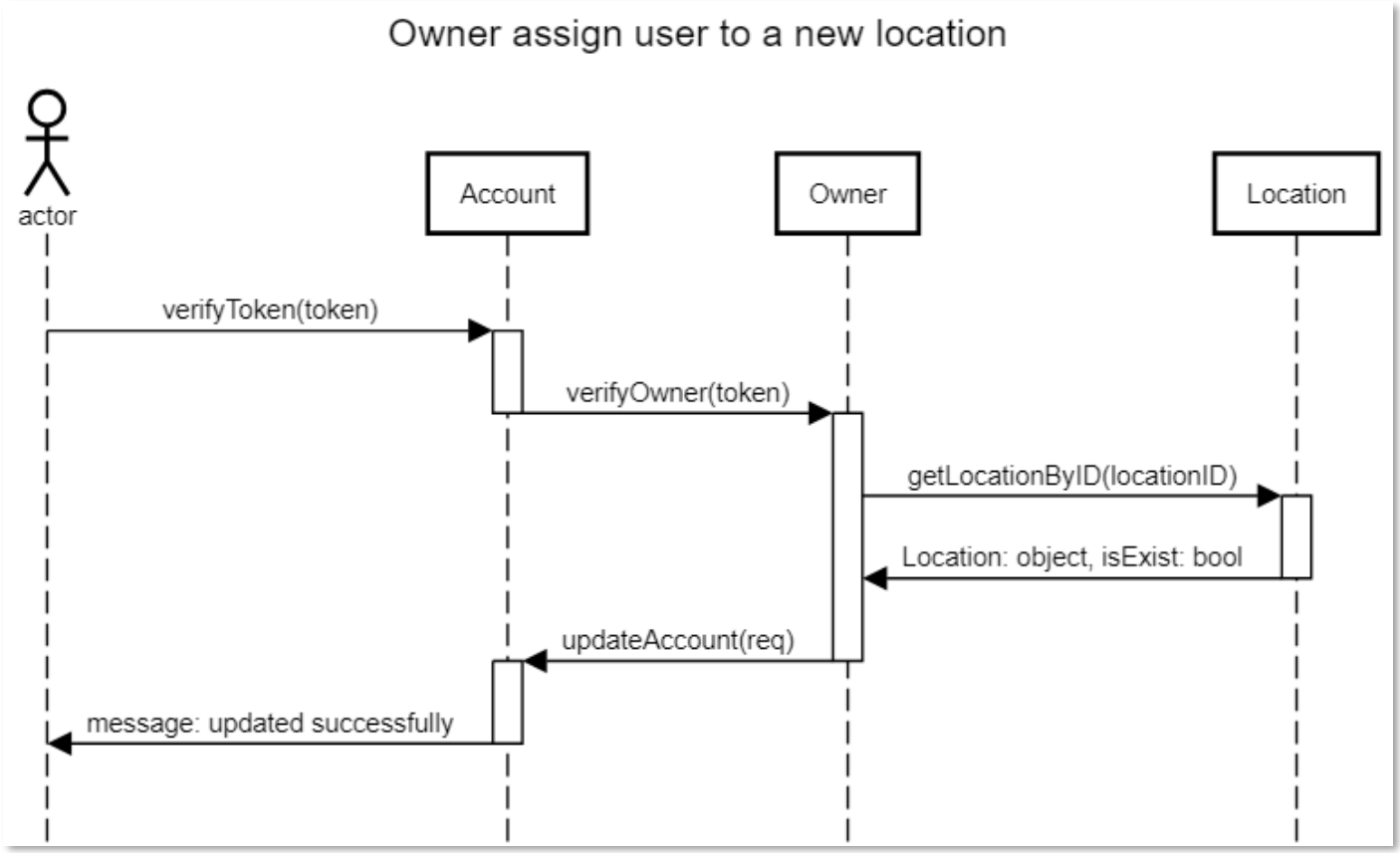

Figure 3: Assign user to a location diagram

To assign a user to a new location, the user first accesses the system, which verifies their token and role. The request is then passed to the owner object, which retrieves the specified location by its ID to ensure it exists. If the location is valid, the owner object updates the user's account with the new location. Finally, a success message is returned to the user.

7.3. Make a receipt from an order

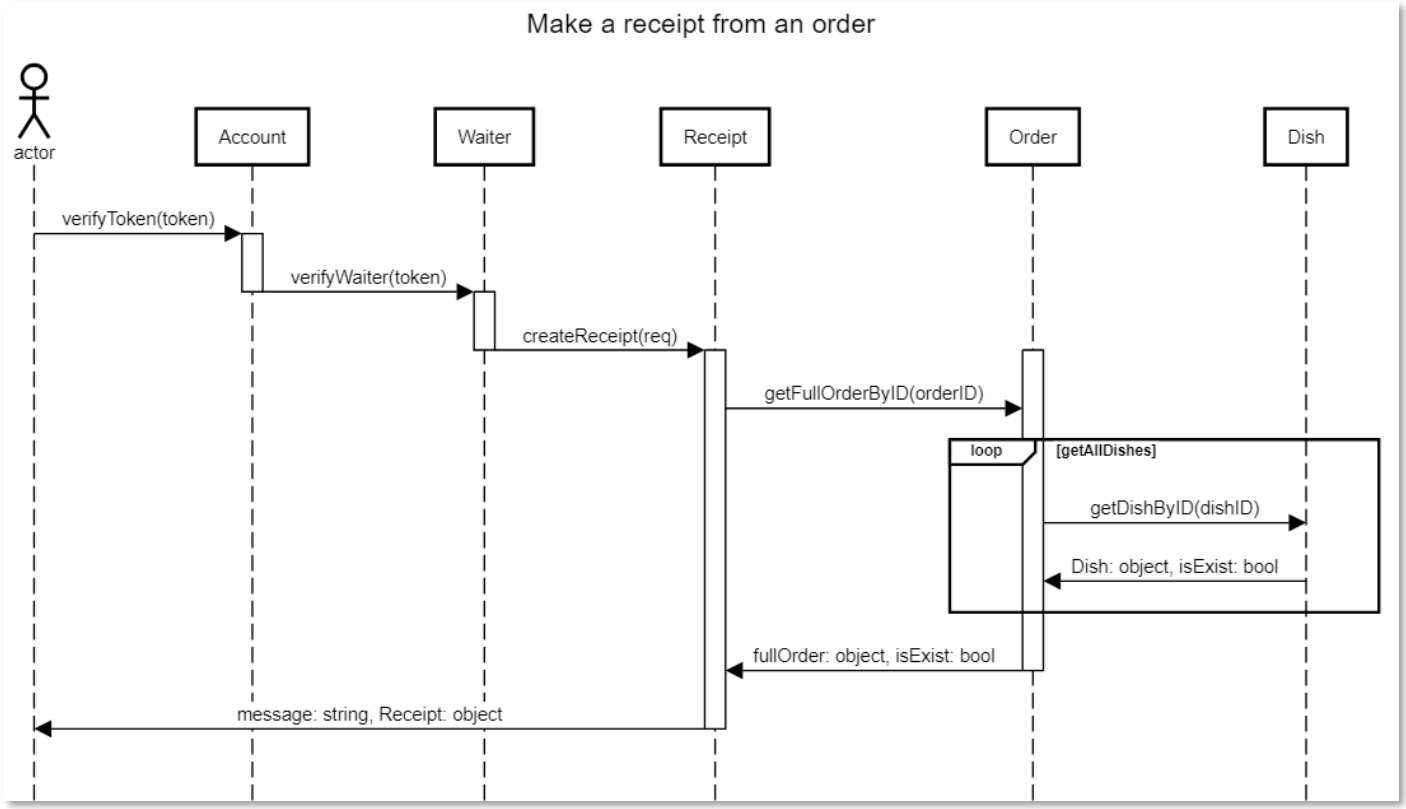

Figure 4: Make a receipt for order

To create a receipt from an order, the user first accesses the system, which verifies their token and role. The request is then passed to the waiter object, which initiates the creation of the receipt. The receipt object retrieves the full order details by its ID, fetching all associated dishes to ensure they exist. Once the complete order is assembled, the receipt is created. Finally, a success message and the receipt object are returned to the user.

7.4. Check for table's availability and make a reservation

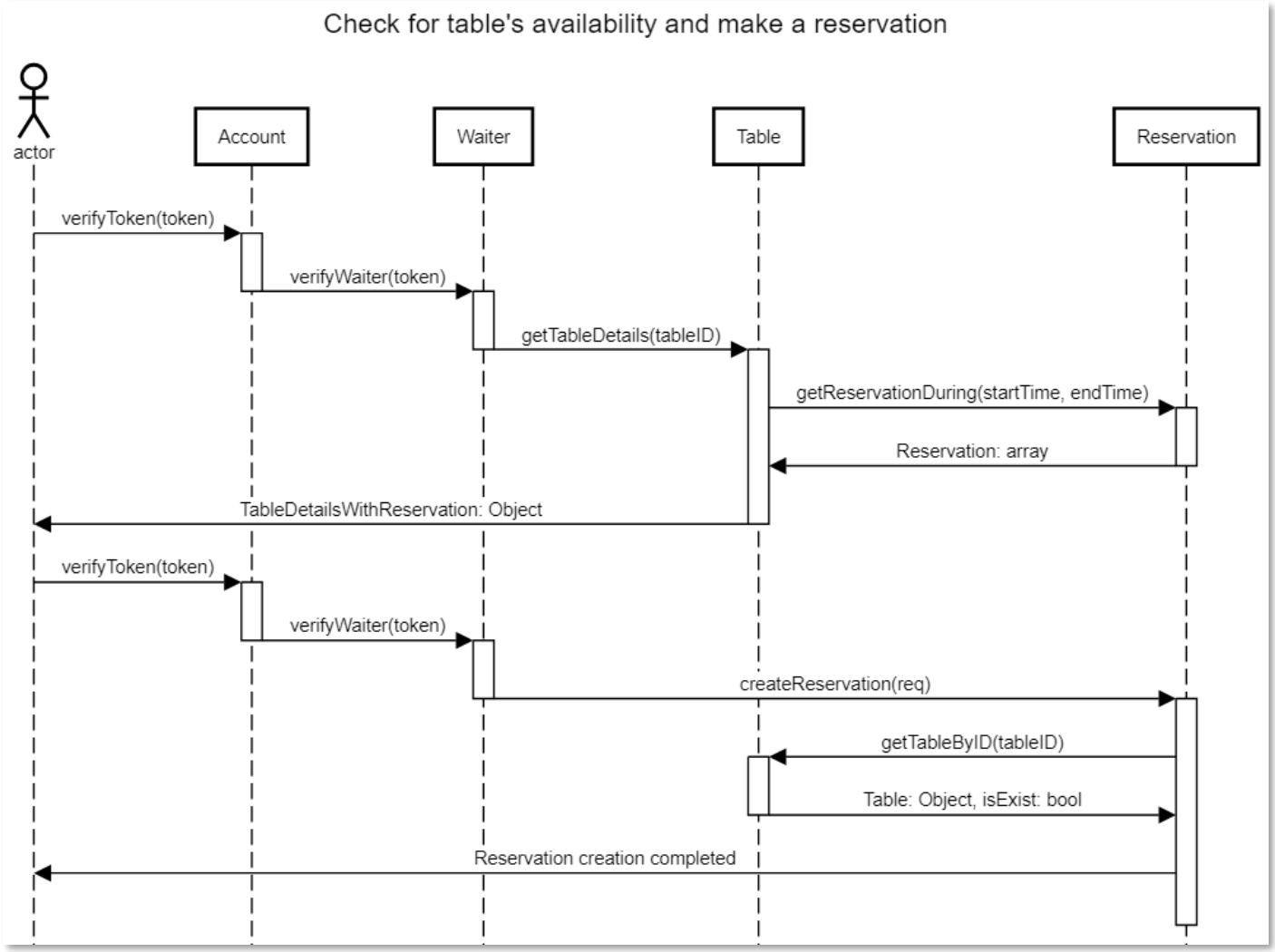

Figure 5: Check for table's availability and make a reservation

To check for a table's availability and make a reservation, the user first accesses the system, which verifies their token and role. The request is passed to the waiter object, which retrieves the table details. The table object then checks for existing reservations during the specified time. If the table is available, the details and reservations are returned to the user. The user then makes another request to verify the token and role, followed by creating the reservation. The waiter object passes the request to the reservation object, which confirms the table's existence and completes the reservation. Finally, a confirmation message is returned to the user.

VIII. References

1) Mozilla. (n.d.). MVC. MDN Web Docs. Retrieved July 14, 2024, from https://developer.mozilla.org/en-US/docs/Glossary/MVC
2) Oracle. (n.d.). Intercepting filter. Oracle Java Technologies. Retrieved July 14, 2024, from https://www.oracle.com/java/technologies/intercepting-filter.html
3) Polito, M. (2020, January 22). REST API design best practices. freeCodeCamp. Retrieved July 14, 2024, from https://www.freecodecamp.org/news/rest-api-design-best-practices-build-a-rest-api/