VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE

OMGVA Technical Design

Team Members:

Gita Gliaudytė,
Mantas Vadopalas,
Vladislavas Malajevas,
Augustinas Jarockis,
Olen Račkauskas

Vilnius
2024

Table of Contents

Introduction

This technical document presents a thorough overview of the architecture of the Point of Sale (PoS) system. It aims to provide detailed information about the system's structure, components, and interactions. The intended audience for this document includes system architects, developers, and other stakeholders involved in the development, and maintenance of the OMGVA PoS system.

The OMGVA PoS system is designed to make service (food and beauty) sector operations more efficient. This system is made to be suitable for every kind of sector or service. It includes handling orders and reservations, inventory management, refunds and discounts logic. It also incorporates user management and sales transactions, as well as the legal part of taxes.

Business flows

Super Admin creation

Super admins are system administrators (IT support from OMGVA PoS system), more information will be provided in later sections.

During the initial setup of the OMGVA PoS system, the first Super Admin account must be created manually. This should be done by the developer through the database. The developer should enter the first Super Admin details, insert a securely hashed password, and assign full permissions (set the user's role to *SUPER_ADMIN*).

When logging into the system, Super Admin should immediately change the password for the account's safety.

Super Admins have access to create any users with any permissions. Thus, if another Super Admin account is needed, it should be made by the Super Admin in the system.

Business creation

Businesses can only be created and edited by the Super Admin. The Super Admin enters business details, such as business name, address, contact phone number, email address and currency. Then, Super Admin can create an account for the owner. If there are multiple owners, the Super Admin must create accounts for each owner with specific permissions in the system (set the user's role to *OWNER*).

Once the owner's account is created, the owner can log in using a password provided by the Super Admin. The owner can change the password later. Owners can add categories and products and create accounts for their employees. These operations also can be performed by the Super Admin.

Login flow (authentication and authorization)

For *authentication*, users will use username and password.

For *authorization*, we will use JWT. When user logs in, their role, user_id and business_id must be encoded in the JWT token. This makes it easy to manage permissions (based on their role) and find only the relevant resources (users can only manage resources of their own business) without needing explicit params in API for business etc. The returned JWT token in login response should then be used as a Bearer token in all further requests (check Authorization header in swagger)

Example token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJ1c2VyX2lkIjoxMjMsImJ1c2luZXNzX2lkIjoxMjMsInJvbGUiOiJPV05FUiJ9.LhNcIVgUS0PFxN-uNRElnTnEC1Um4Ubep5hq93HmDLc

Can check decoded value in https://jwt.io/

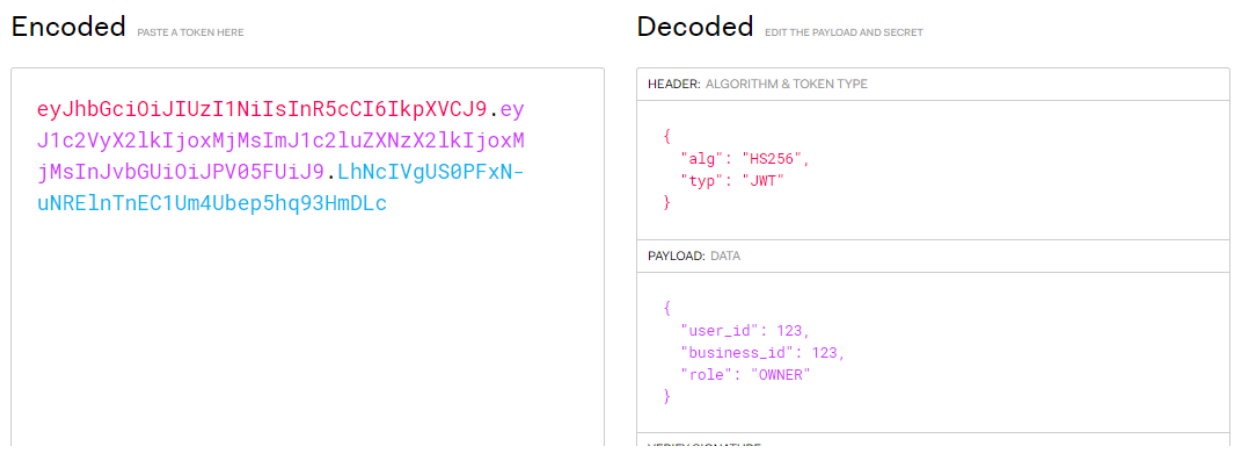

Figure 1: JWT token decoding. Sourced from: https://jwt.io/

Introduction to JWTs https://jwt.io/introduction
(Implementation details how to create/use them will be different based on programming language, however principles are the same in all of them)

Flow will be something like this:

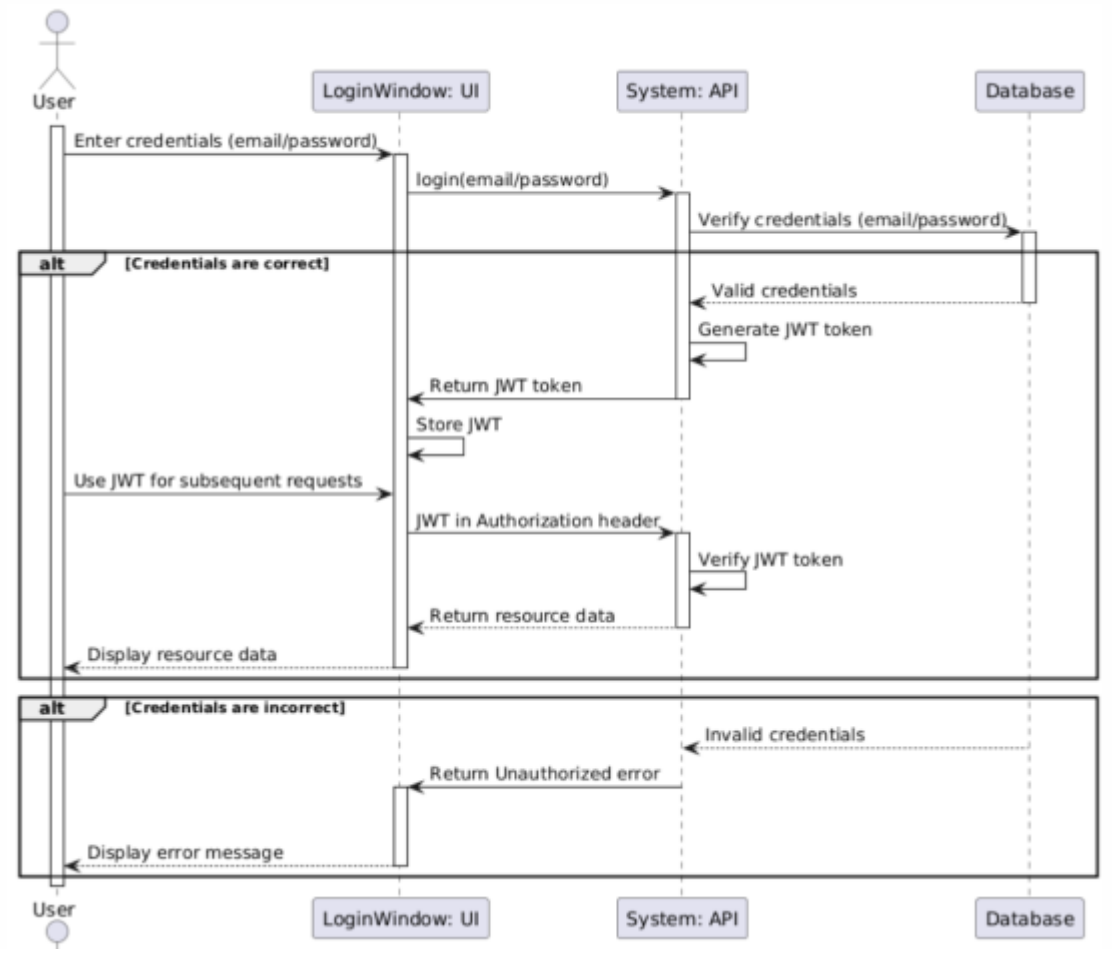

Figure 2: Login sequence diagram. Made with: www.plantuml.com

Note that JWT token is not stored in the database. JWT token is stateless. Do not forget to add expiration to JWT token. One example would be to use refresh token endpoint once it's about to expire.

Ordering products flow

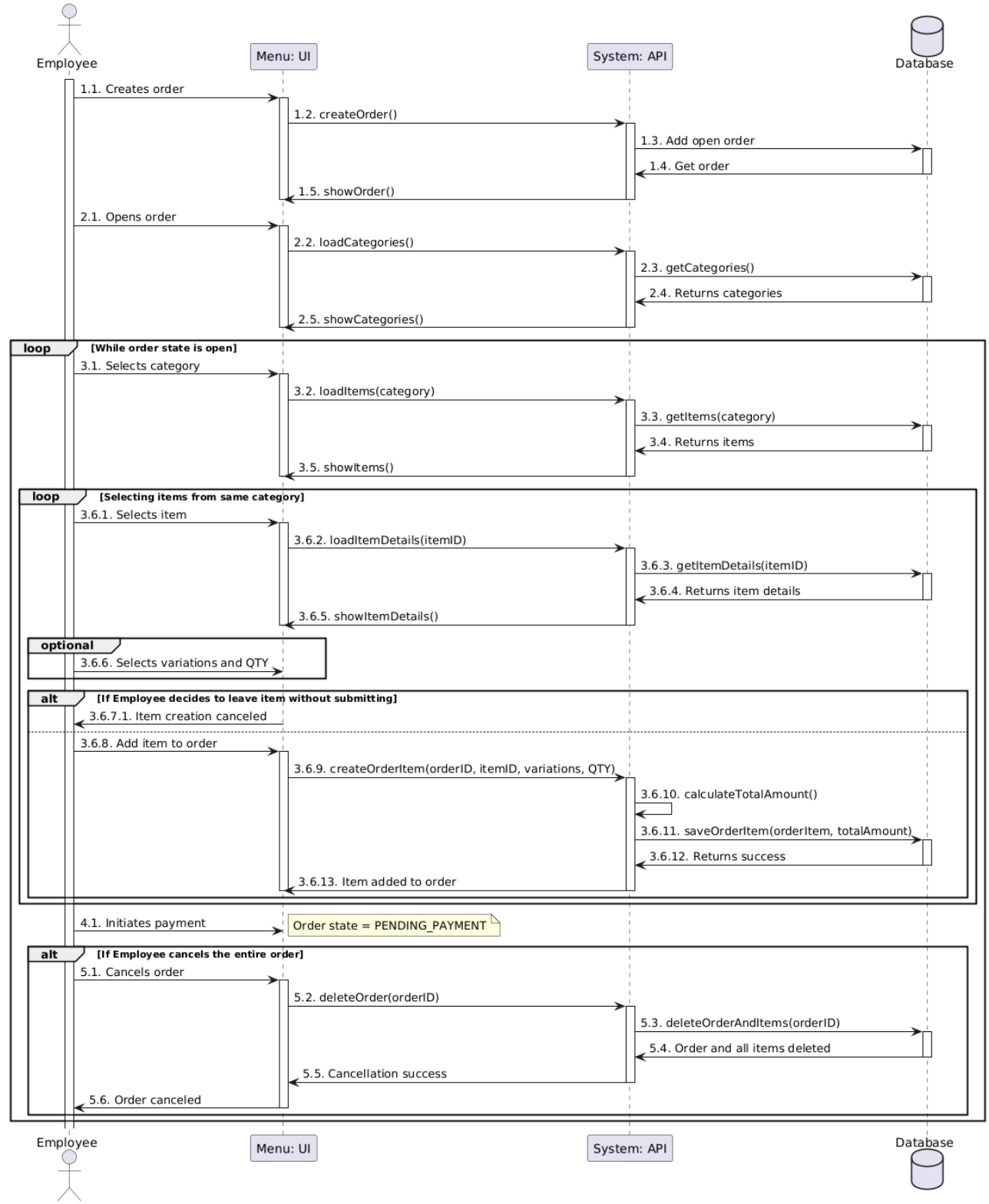

Figure 3: Order sequence diagram. Made with: www.plantuml.com

The process of creating and managing an order process is showcased in the sequence diagrams (see Figure 3, Figure 4).

The employee creates an empty order, which adds the order to the database with an *OPEN* status. The system retrieves the empty order and loads categories for an employee. Next, the employee will put the customer's order through the menu: the employee selects a category, prompting the system to retrieve and display the relevant and available items. When selecting an item, the employee can view its details, choose variations and specify the quantity. If an employee decides to add the product to the order, the system will create a new order item and recalculate the order's amounts of taxes, discounts and the total amount of the order. If the employee chooses to cancel the item addition, the system does not modify the order.

Once all items are added, the employee initiates the payment process, changing the order status to *PENDING_PAYMENT* and signalling that the order is ready for finalization. If an order needs to be cancelled, then it is deleted from the database with all its order items and applied discounts.

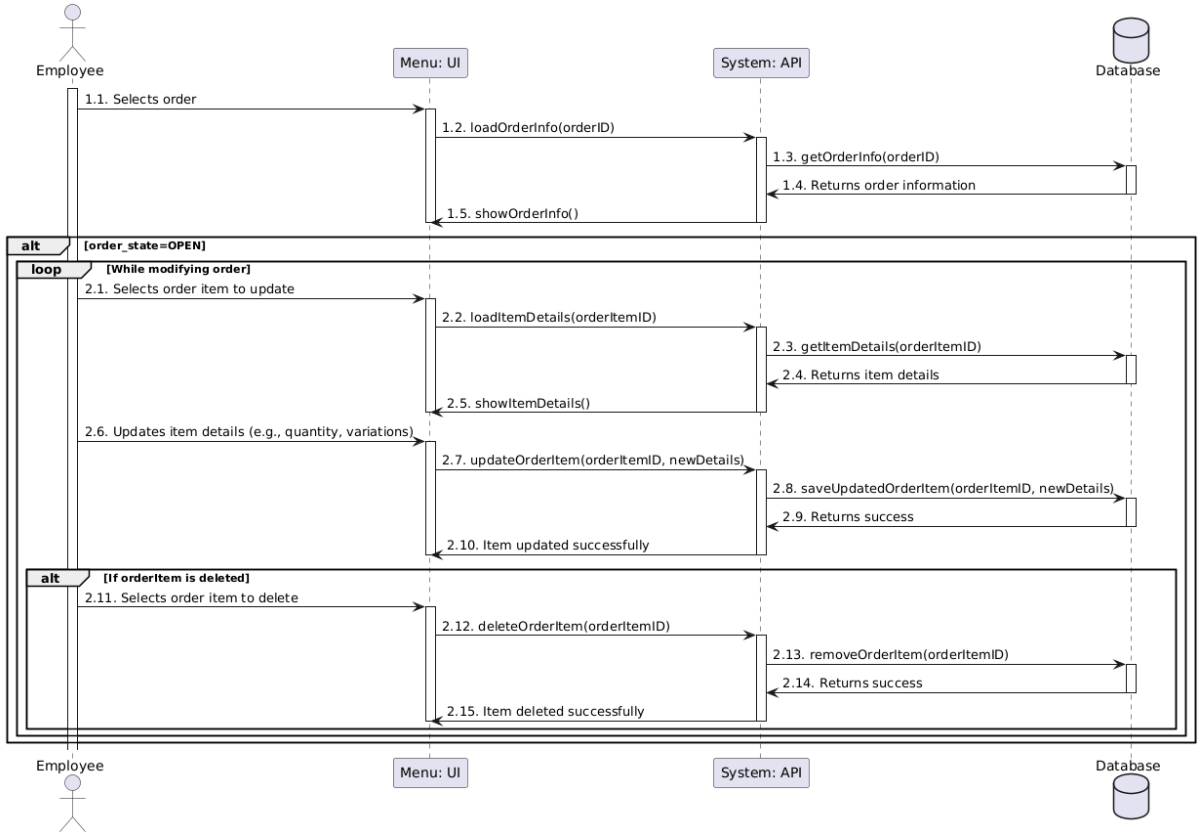

Figure 4: Order modification sequence diagram. Made with: www.plantuml.com

The items in the order can be modified only when the order status is *OPEN*. The employee then selects the item that needs to be updated, and the system returns and displays details of it. The employee then can update the quantity and variations of the selected item. An updated order item is saved to the database, and order amounts are recalculated accordingly. If the order item needs to be deleted, the system will remove the specified item from the database and recalculate the order amounts.

Reservation flow

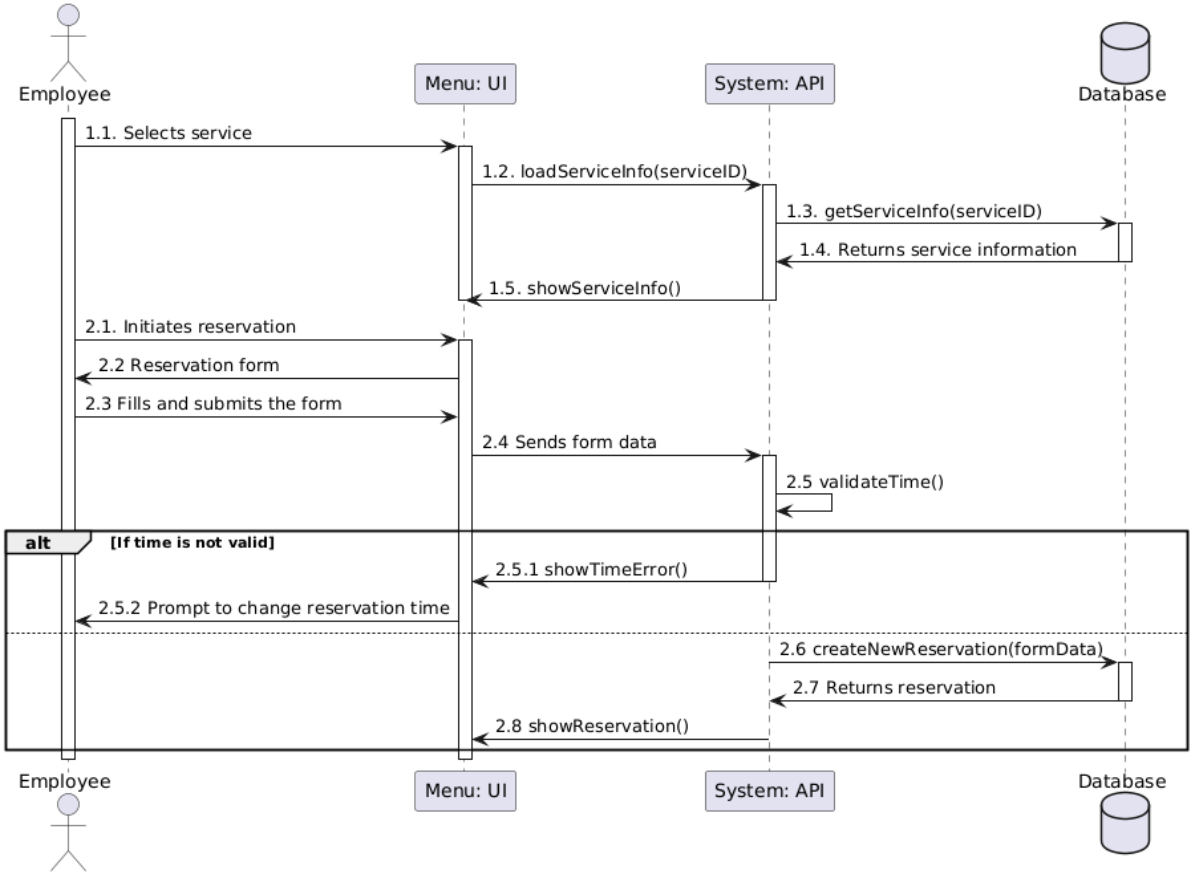

Figure 5: Reservation sequence diagram. Made with: www.plantuml.com

Reservation flow is technically an additional modification of the order item, whose type is *SERVICE*. This means that the reservation can only be made after adding a service to an order.

The process itself starts with selecting the service (that was added to the order). The system returns and displays the service information to the employee. The employee initiates the reservation by clicking a button. Then, the UI will display the form, which is filled in and submitted by the employee. The form data is sent for validation. If the reservation time is not valid (e.g. time is already occupied), the prompt to change the reservation time is displayed. If the time is valid, then a new reservation is saved to the database and showcased to the employee. Modification of the reservation works the same way, but instead of creating a new reservation, the older one will be updated.

Discount flow

Discount has *valid_from* date, *valid_until* date and *discount_hash* (all are nullable).

1. *valid_from* indicates from which date the discount for the product can be applied (if not specified, discount doesn't have a start date)
2. *valid_until* indicates the last date that the discount for the product can be applied (if not specified, discount doesn't have an expiration date)
3. *discount_hash* is the "coupon code". If *discount_hash* is null, customer does not need to provide a coupon code (discount is applied based on previous rules). If *discount_hash* is not null, customer must provide the coupon code

Example code, that can be used as reference to check whether discount can be applied.

```
boolean canBeApplied() {
  return
     (valid_from == null      || currentDate.after(valid_from))
  && (valid_until == null     || currentDate.before(validUntil))
  && (discount_hash == null   || couponCode.equals(discount_hash))
}
```

Figure 6: Code of a function that checks if a discount can be applied.

We have two tables for discounts. One is ‘Discount’ table, that configures discounts. And one is ‘AppliedDiscounts’ which stores the actual applied discount amount for order item.

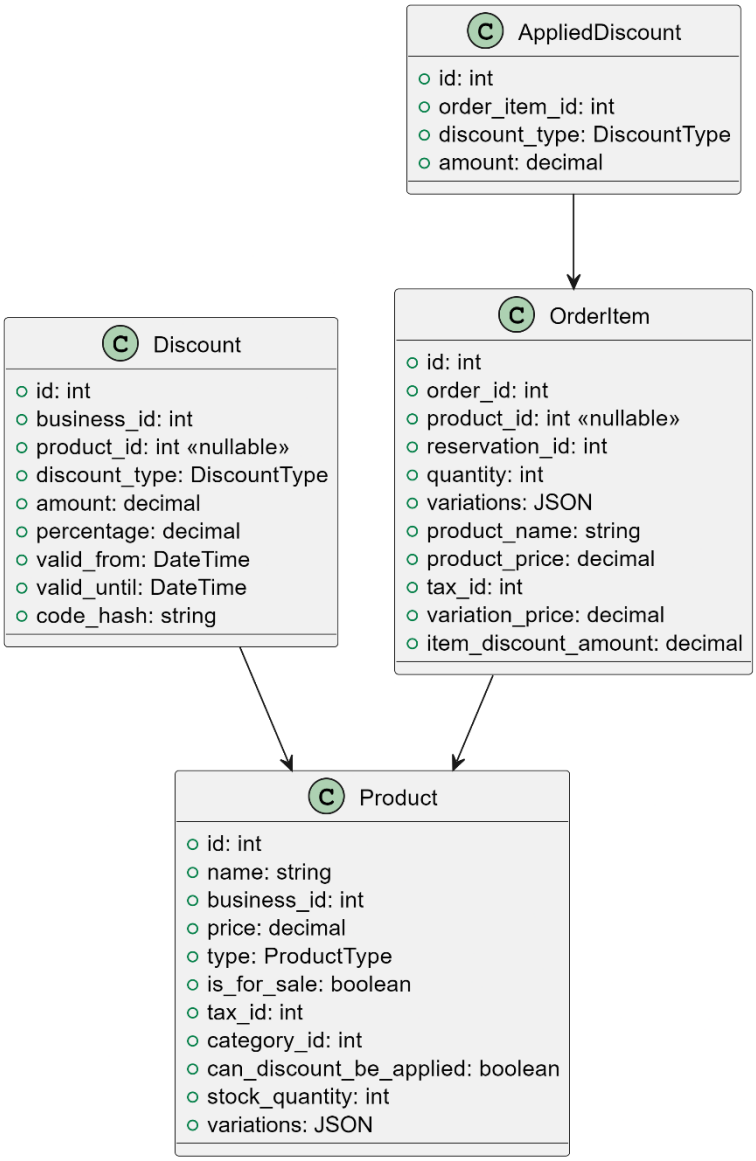

Figure 7: Class diagram of entities related to discounts. Made with: www.plantuml.com

How to configure PRODUCT discount:

- Set the product_id of the product that you want discounted
- Set discount_type as PRODUCT
- Set the amount to indicate what should be the discount OR set the percentage that should be discounted
- Configure valid_from, valid_until and code_hash (all are nullable)

- Constraints (for product discount)
    - Product_id is mandatory
    - Amount or percentage is set

How to configure ORDER discount

- Set *discount_type* as ORDER
- Set the percentage of the discount
- Configure *valid_from*, *valid_until* (both are nullable)
- Configure *code_hash* (required)
- Constraints (for order discount)
    - *product_id* must be null
    - *code_hash* is required
    - Amount is null (order discounts are applied based on percentage)

Because some products (alchohol in Lithuania for example) cannot be discounted, in product table there is a flag *can_be_discounted*. If this flag is false, neither order item or order discount can be applied for that product (default is true).

There are two ways discounts can be applied. One is when adding product to order. And another is when order items are already present, and we apply some coupon.

Constraints – order item can have up to two Applied Discounts

- At most one ORDER_ITEM discount
- At most one ORDER discount

Order Item discount when adding product to order

When adding a product to an order, the system should always try to apply a discount. This makes sure, that discounts that don't need a coupon code are always applied (if they satisfy all other conditions).

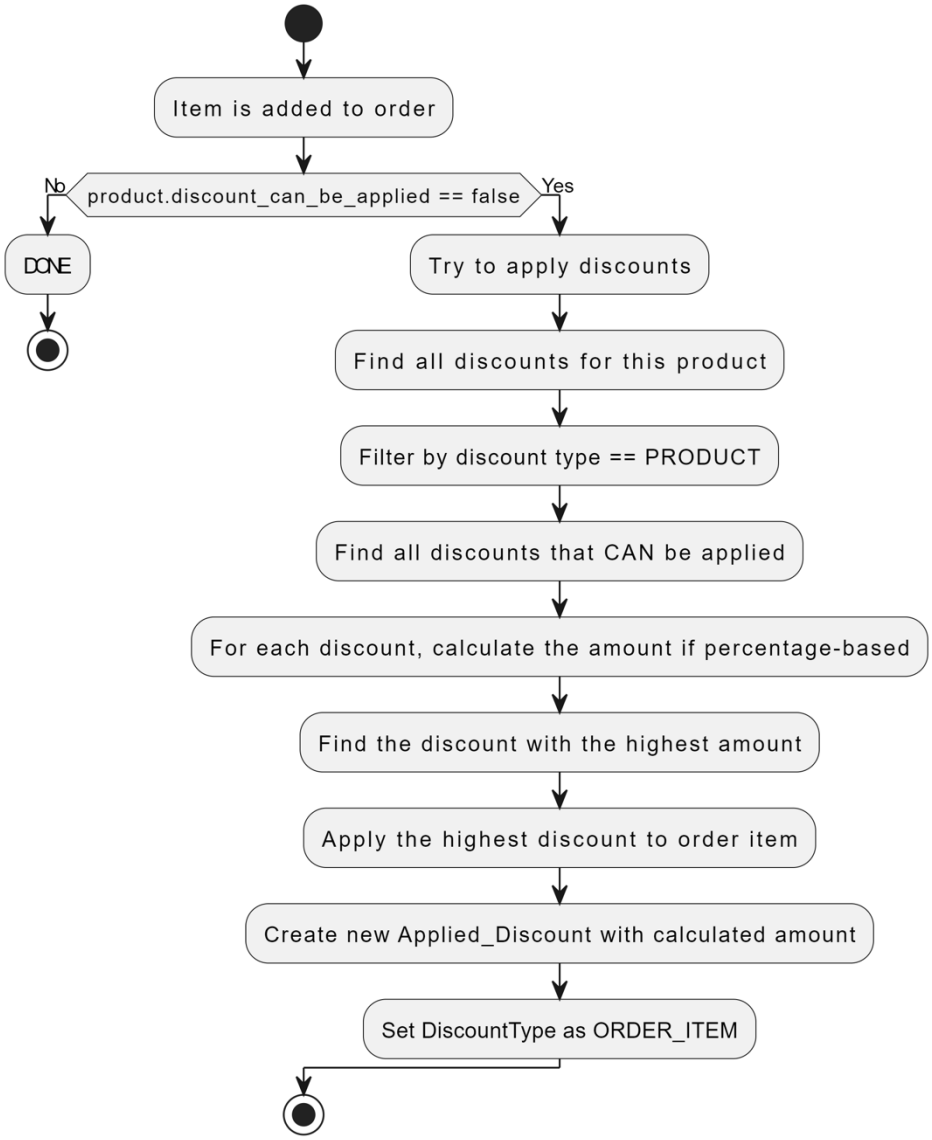

Figure 8: Application of discounts on addition of order_item. Made with: www.plantuml.com

Order Item or Order discounts when applying a coupon

A coupon code discount can be applied as an order item discount (specific for one product) or as a discount for the whole order.

To make sure that employees can't just apply a 100% discount on everything, the owner needs to configure order discounts that may be applied, and they can only be applied by specifying the order coupon code. We cannot just set the percentage directly.

Full discount flow can be checked in attached plantUML diagrams, here showing just the snippets.

First, we check whether the coupon code is even valid.

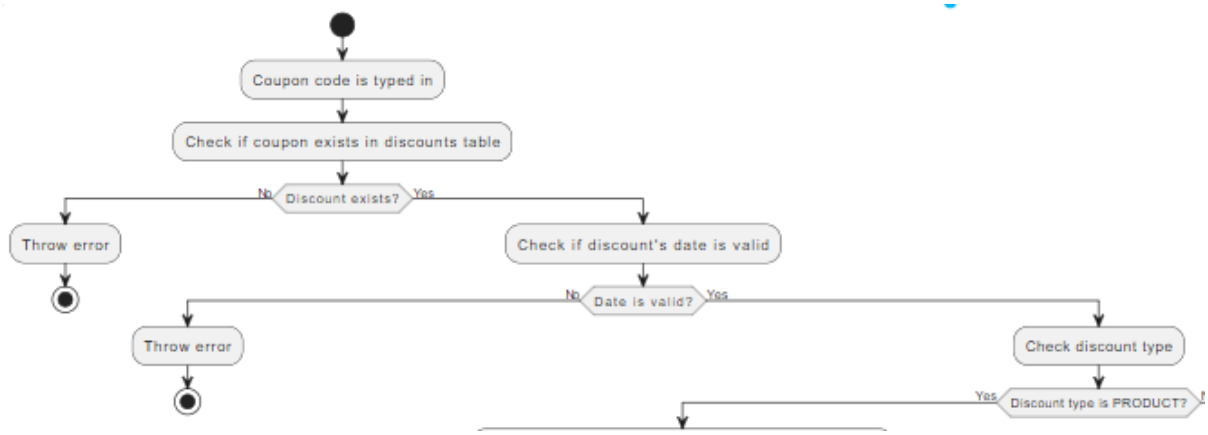

Figure 9: Flow diagram of coupon validation. Made with: www.plantuml.com

Then, based on whether the discount is for PRODUCT or for ORDER we execute different logic.

For product discount, we first check if discount (of type ORDER_ITEM) is already applied for it and keep the one with higher discount amount.

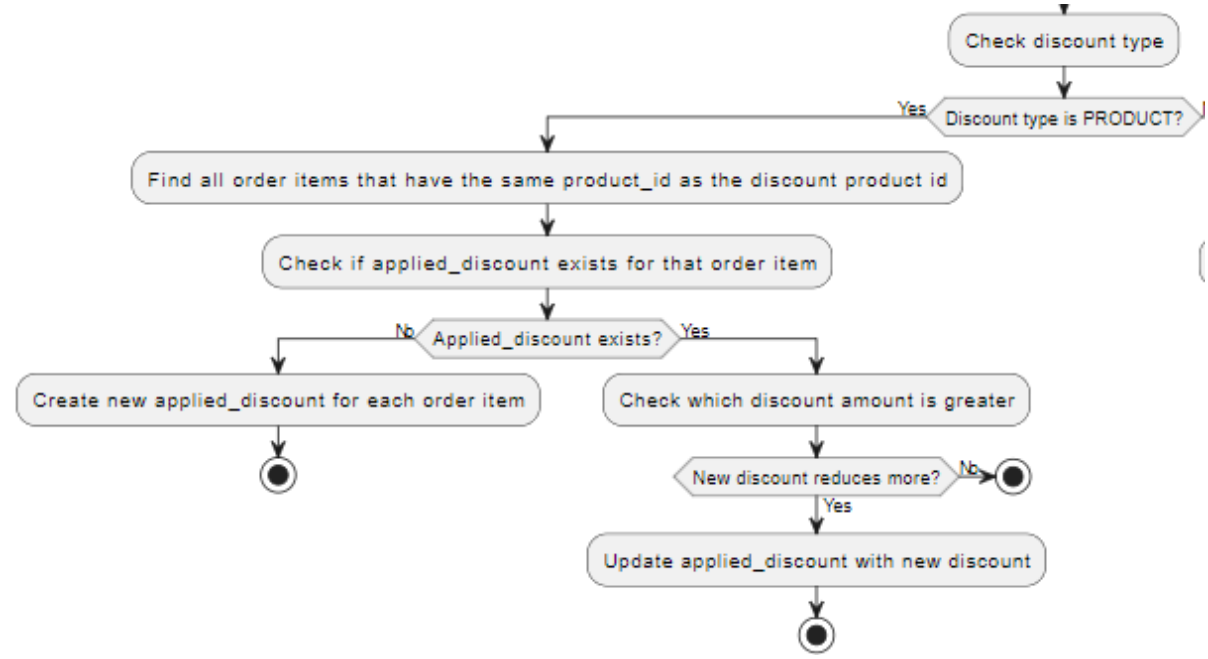

Figure 10: Product discount flow diagram. Made with: www.plantuml.com

For order discount we keep a percentage. If the new order discount has a lower percentage, we still use the new one to make sure that order discounts can be reduced, if an employee accidentally inputs a wrong discount.

Creating the applied discounts has the same flow as when adding product to order.

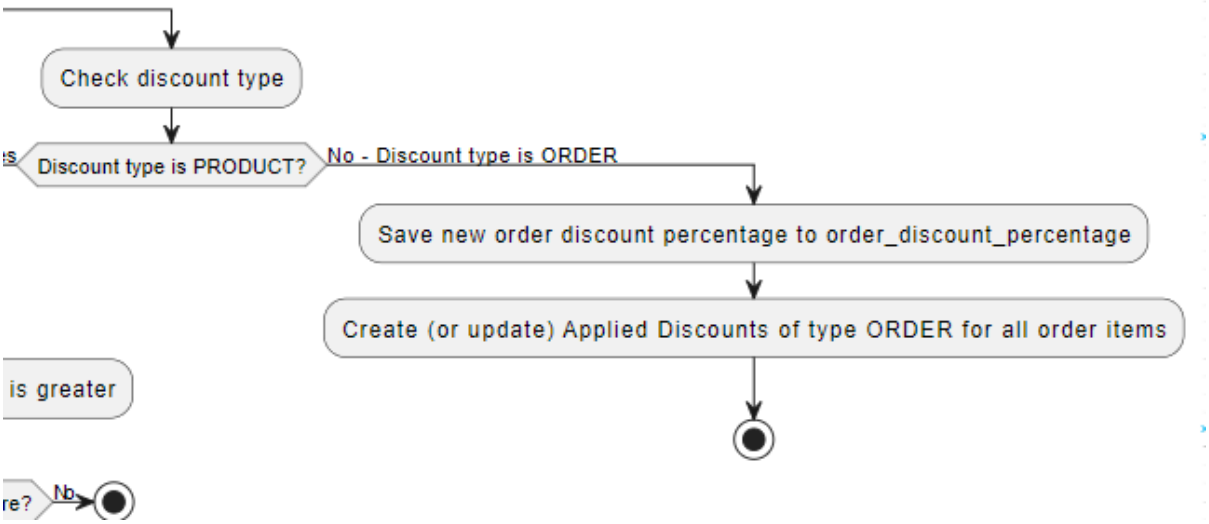

Figure 11: Order type discount flow diagram. Made with: www.plantuml.com

Payment flow

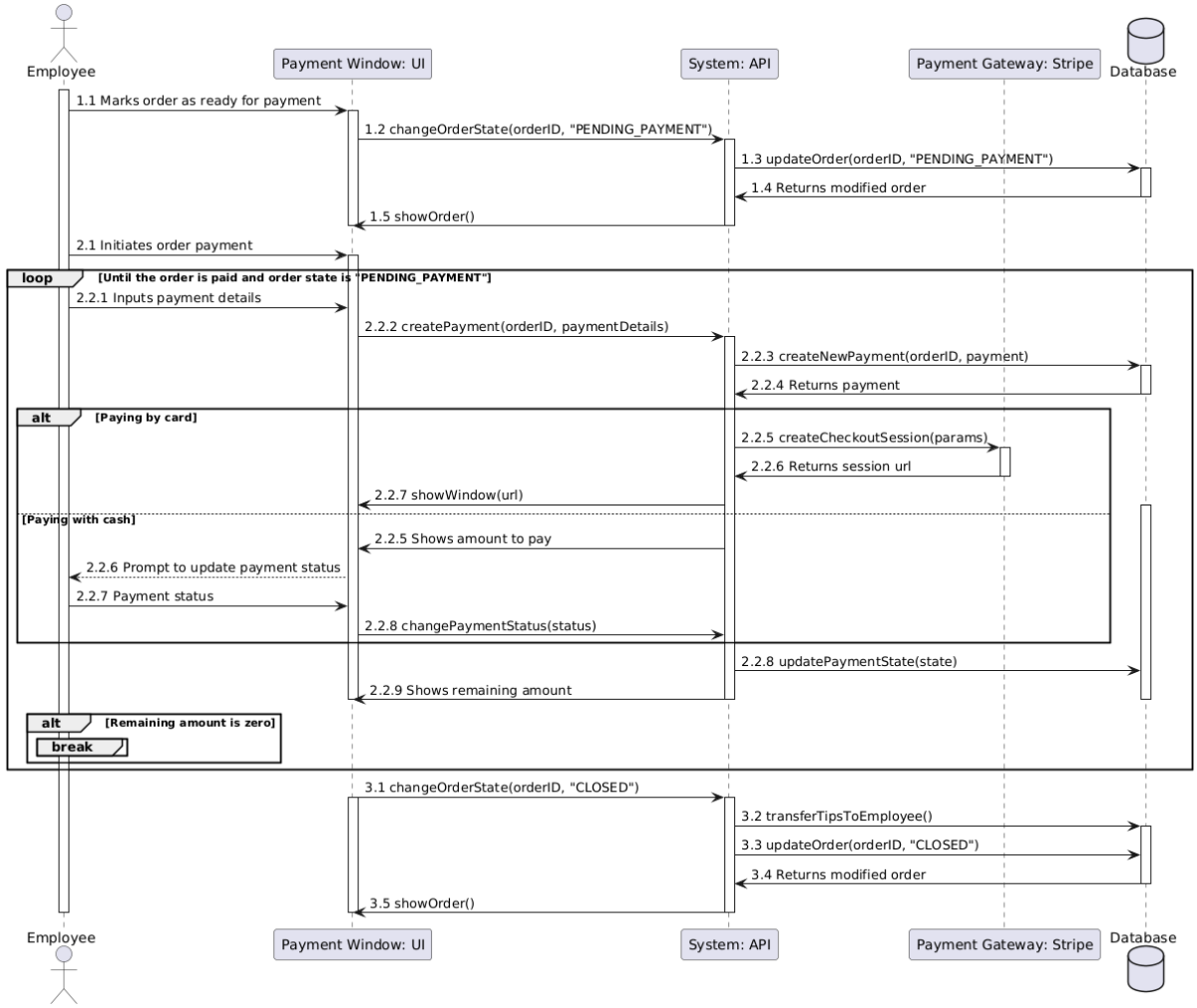

Figure 12: Payment sequence diagram. Made with: www.plantuml.com

The payment process varies based on the chosen payment method. The sequence diagram (see Figure 12) illustrates the payment process for cash or card. First, the employee marks the order as prepared for payment, changing the order status to *PENDING_PAYMENT*. Then, the employee enters the payment details (payment method, tip amount, and total amount) to create a new payment with the status *PENDING*.

If the payment method is *CARD*, the amount and currency are sent to the third-party system, Stripe, which handles the payment process and provides a user interface for it. Following a successful or failed payment, Stripe returns a URL to the designated window.

If the payment method is *CASH*, the user interface displays the total amount due and prompts the employee to confirm or reject the payment (based on whether the correct

amount of cash was received). The payment status will be updated based on the success or failure of the payment, and the remaining amount will be shown to the employee.

For split payments, these processes continue until the remaining amount is zero.

When the order is fully paid, its status will be changed to *CLOSED*, and the tips provided by the client will be transferred to the employee. The paid order will be displayed on the employee's device.

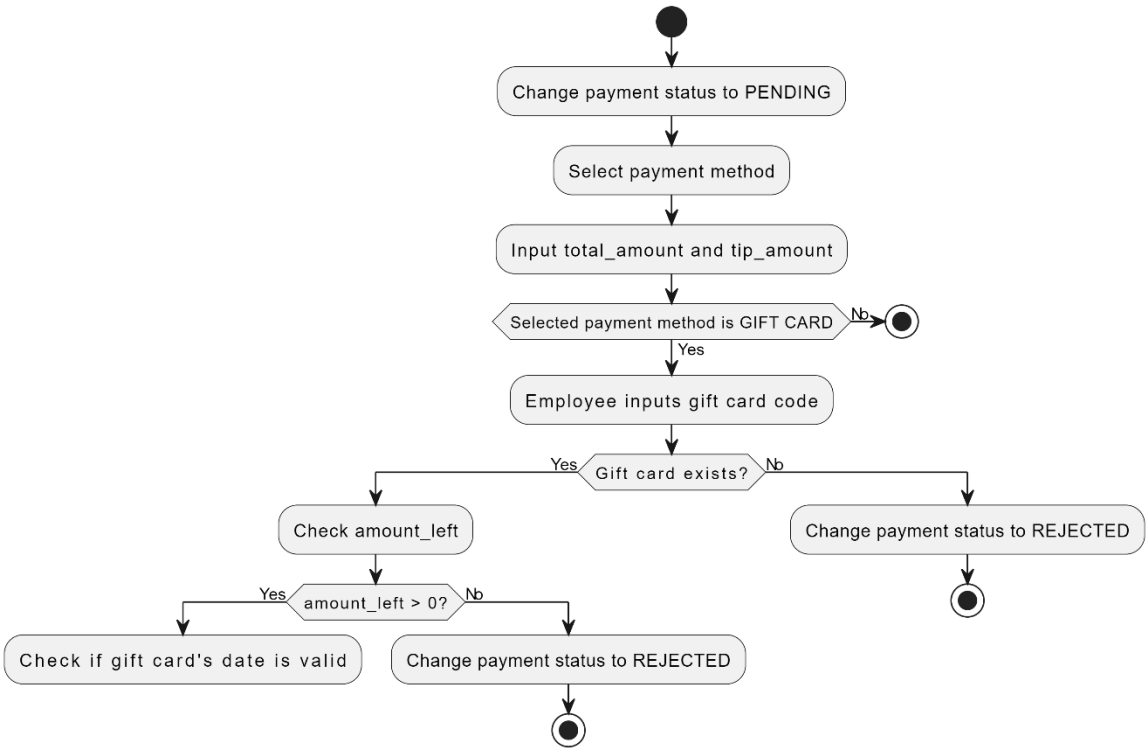

Figure 13: Payment by gift card flow diagram. Made with: www.plantuml.com

The payment process using a gift card is explained in the following flow diagrams. The process begins by changing the payment status to PENDING, creating a payment with the selected payment method as GIFT_CARD. The employee then inputs the gift card code. The system checks if the gift card with the provided code exists. If the card does not exist, the payment is rejected, and the status of that payment changes to *REJECTED*. If the gift card exists, the system checks the remaining balance (*amount_left*). If the balance is greater than zero, the system verifies the expiration date to check the validity of the gift card.

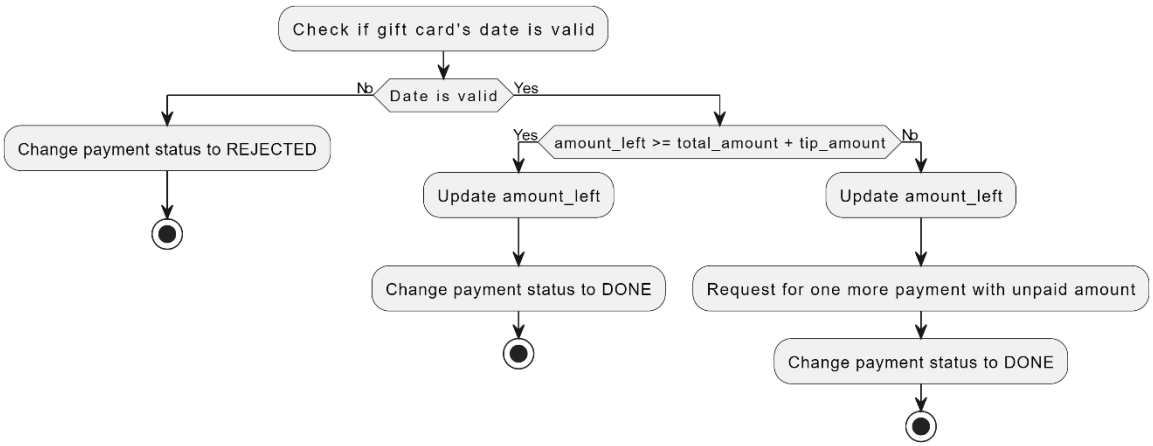

Figure 14: Payment by gift card flow diagram 2. Made with: www.plantuml.com

If the card is invalid, the payment is rejected. Otherwise, the system compares the available balance to the total amount (including tips). If the balance covers the full amount, the remaining balance is updated, and the payment status is marked as *DONE*. However, if the balance is insufficient, the system prompts an additional payment for the unpaid amount.

Refund flow

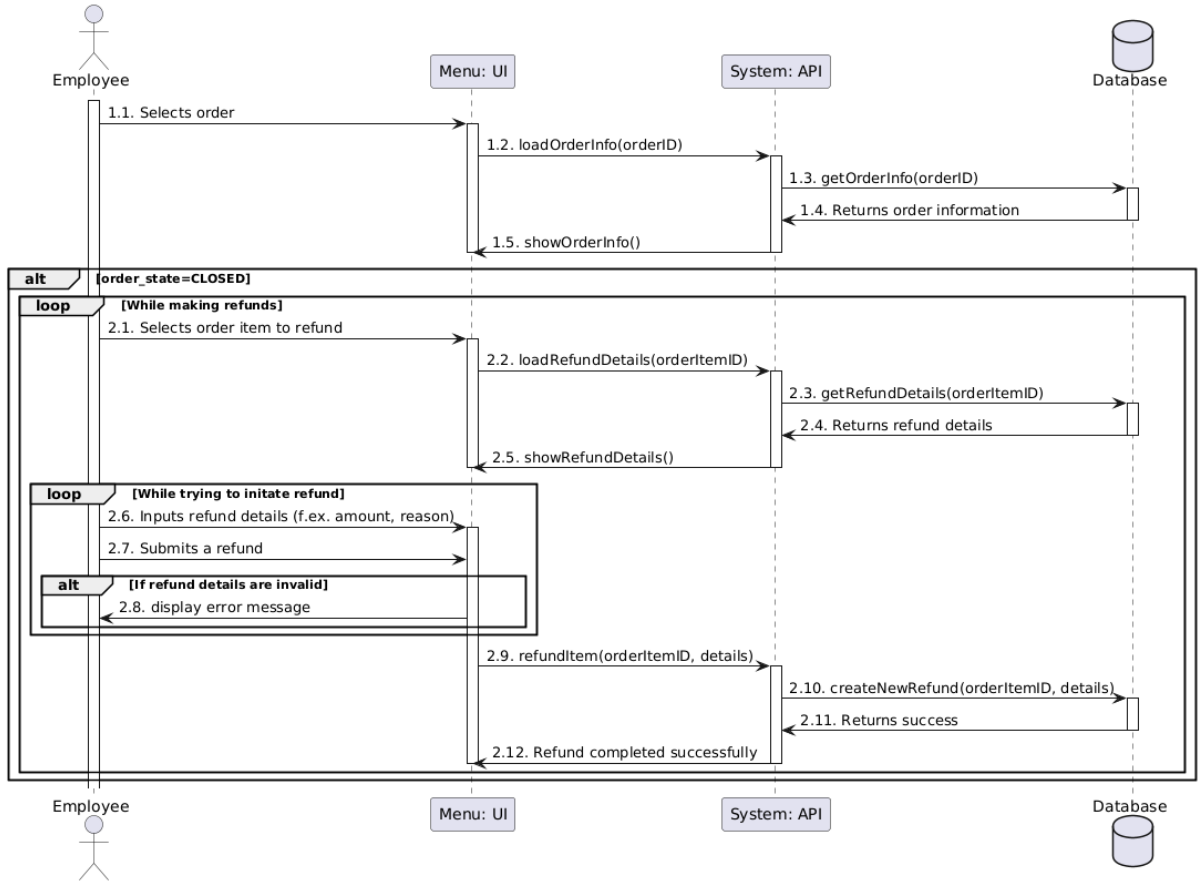

Figure 15: Refund sequence diagram. Made with: www.plantuml.com

Refunds are only possible on orders that are fully paid. So, to be able to initiate a refund, employees must first go into a *CLOSED* (fully paid) order.

Refunds are only done for one *order_item* at a time.

When selecting an order item to refund (2.1.), an API call (2.2.) is made for the respective *order_item*, to get info on whether there were any refunds done for this *order_item* before. This is done, since refund information doesn't come with the order info (1.4. / 1.5.)

Employee then must input (2.6.):

1. If the refunded item will be returned to inventory (E.g.: is the returned cup broken or not),
2. How much of that item is refunded,
3. Amount of money to be returned to the customer for the refund.
4. The reason for the refund

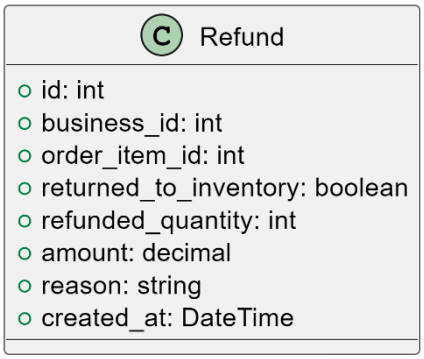

Figure 16: Refund class. Made with: www.plantuml.com

Checks are at the UI level once the employee tries to submit a refund (2.7.) to check if the inputs are consistent with refund details (2.5.). E.g.: 5 cups were bought and according to previous refunds – 2 cups were already refunded some time earlier. If the employee tries to submit 5 cups for the *refunded_quantity*, the employee then is shown an error message (2.8.) so that they enter the correct amount.

If everything is correct, the employee can initiate a refund (2.8.)

Another check is then performed at the API level (not shown in diagram), to check if the refund details are correct. If they (details) are correct – a *refund* is added to the database.

Notes:

- A new refund doesn't affect/changed orders or *order_items* in the database, therefore it does not have *order_id* as one of the fields.
- *amount* means how much money is returned for the refunded items. *amount* is arbitrary (decided by the employee), but it cannot be higher than what the customer has paid initially for the *order_item*

Inventory management flow

Inventory *(stock_quantity)* is saved inside a product. If a product does not require inventory (example - services / meals) *stock_quantity* is null. If product requires inventory, *stock_quantity* is an integer and cannot be less than 0.

Besides manually setting *stock_quantity* with product API, there are 3 main ways inventory can be changed.

1. When adding product to order (or updating order item quantity in order)

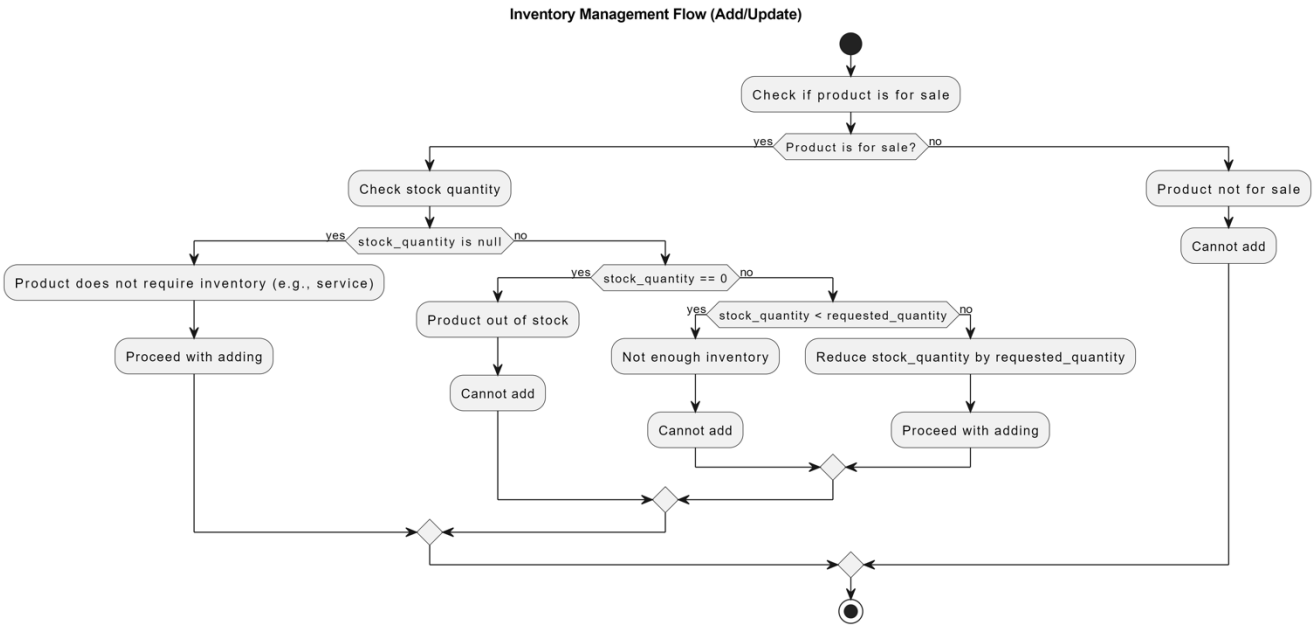

Figure 17: Inventory management flow diagram. Made with: www.plantuml.com.

2. When removing order item from order

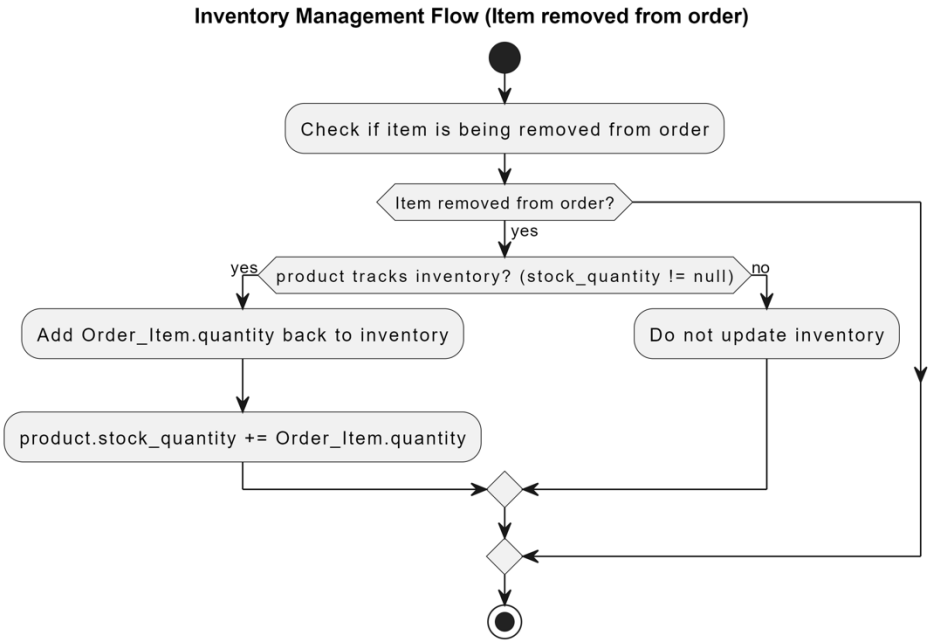

Figure 18: Inventory management flow diagram for item removal. Made with: www.plantuml.com

3. When refunding order item (keep in mind, that item may not necessarily be added back to inventory. For example, if product is damaged, *returned_to_inventory* will be false)

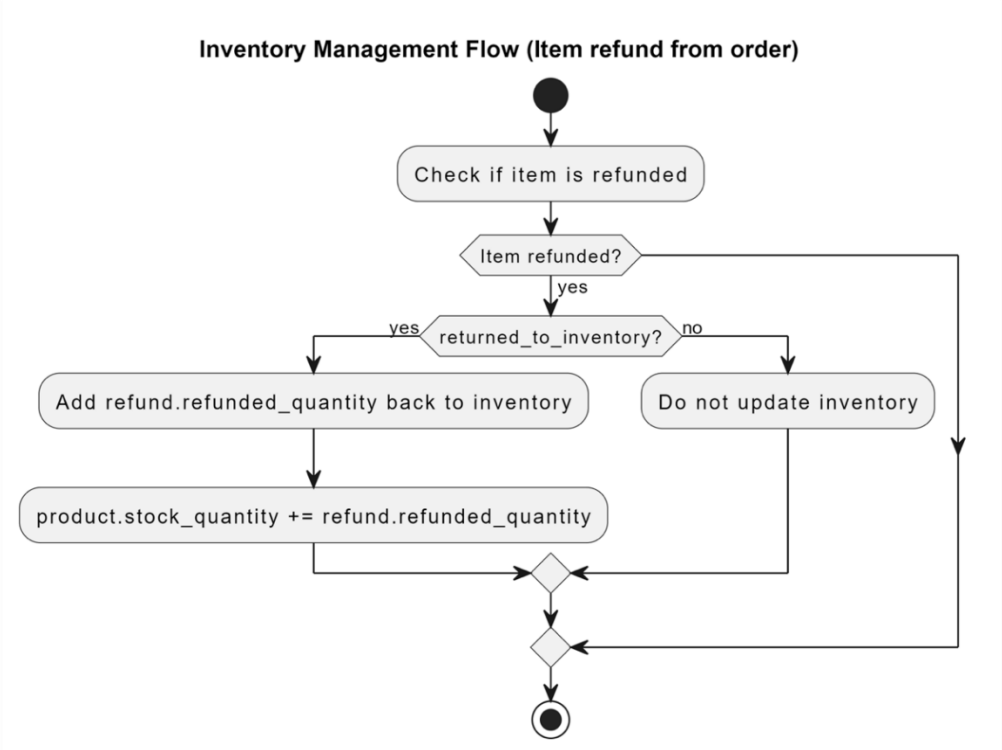

Figure 19: Inventory management flow diagram in case of refund. Made with: www.plantuml.com

Tax management

In the OMGVA PoS system, tax management is designed to ensure that existing taxes cannot be deleted once created. Instead, taxes can either be created or *updated*. When a tax is updated, the system does not directly modify the existing tax. Instead, it creates a new tax entry with the updated *tax_rate* and marks the older version of the tax as invalid (setting *is_valid* to false). This approach maintains a history of tax changes.

The association of products with tax is automatically updated to the newly created tax entry, ensuring that future transactions use the updated tax rate. However, for existing order items, the system retains the tax of the older version.

Additionally, only a Super Admin can add or update taxes. The system should include the base taxes, and if there is a business requirement for new taxes, the Super Admin will be responsible for adding them. Also, there are a requirement for Super Admin to name taxes like so: <*Country Code*>_<*Tax_Name*> (e.g. LT_PVM_Alcohol).

High-level architecture

Package diagram

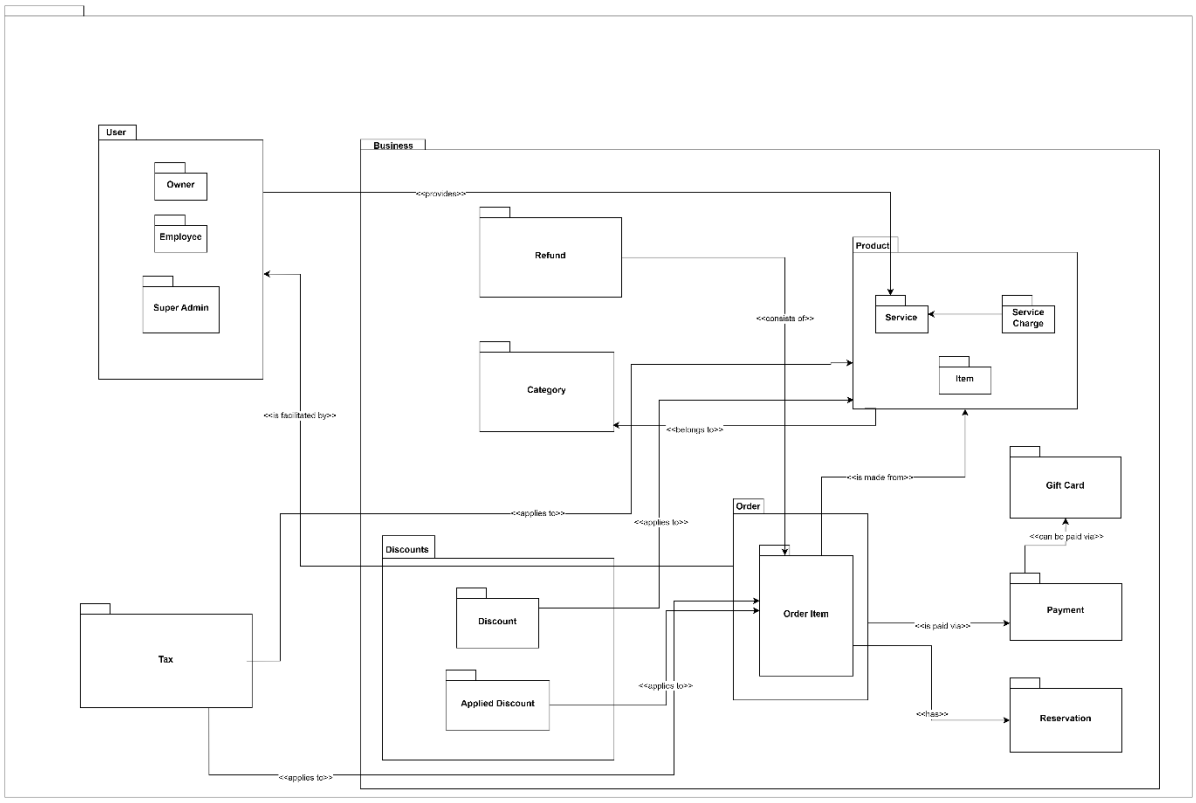

Figure 20: Package diagram. Made with: www.plantuml.com

The package diagram illustrates the high-level architecture of the OMGVA PoS system (see Figure 20). The *Business* package serves as the core, encompassing almost all the system's components. Within this package, there is a *Product* package which contains business offerings such as *Service*, *Service Charge*, and *Item*. These offerings are grouped into *Categories*. Orders are processed within the *Order* package, which includes the *Order Item* package. Order Items are created from the *Product* and can have a *Reservation*. Payments for Orders are managed through the *Payment* package. Additionally, the *Gift Card* allows for orders to be paid via gift cards, providing customers with another flexible payment option. *Discounts* and *Taxes* are applied to *Products* and *Order Items*. *Refunds* are handled separately to facilitate returns. Lastly, the *User* package comprises the *Owner*, *Employee*, and *Super Admin*, who interact with the *Business*.

Data Model

The data model of the OMGVA PoS system is showcased in three class diagrams.

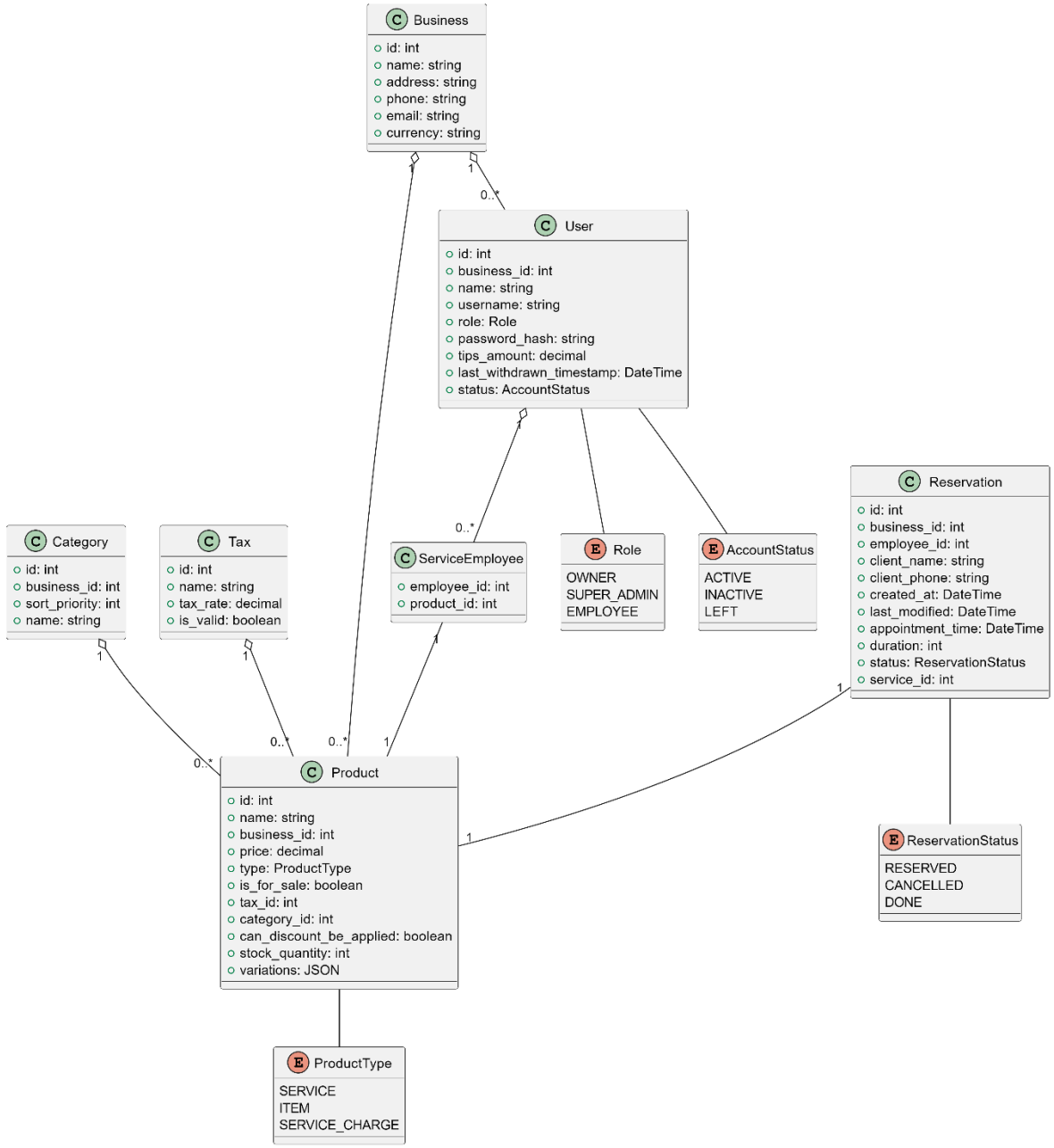

Figure 21: Data model for everything excluding the order part of the business. Made with: www.plantuml.com

The first diagram (Figure 21) represents the business excluding orders part of the data model. Business class is the core entity of this data model. Each Business has their own Categories, Products, Discounts, Users, Reservations, Payments, Refunds and Orders (orders will be visible in the following diagram). Each business has its Users with at least one Owner. The User class belongs to the business as an employee. User can login to the system and interact with it. Each User has its role in the system:

- A *SUPER ADMIN* can do everything as any business owner and they can create and modify businesses, add and edit taxes, create new Super Admins.
- An *OWNER* can do everything within a business except for editing CLOSED orders or altering business information.
- An *EMPLOYEE* can create orders, reservations, and payments and organize refunds.

The User (Owners and Employees) can collect and withdraw tips from the orders they make.

The Product belongs to the business. It has three types: *SERVICE, ITEM* and *SERVICE_CHARGE.* The Product can be marked as for sale (can be offered to customers) or not for sale (cannot be offered to customers or the quantity is zero). The inventory of products is held within *stock_quantity*.

Products belong to categories which are business specific and have their own sort priority. They also have a tax rate. A product may be indicated to have a discount.

Each Employee or Owner can perform services (e.g. a specific kind of massage). This relationship is described in the *ServiceEmployee* table (e.g. employee with an *employee_id 33* can do a *SERVICE* with *product_id* 115). If Product is a type of service, then a Reservation may be created. A Reservation has an employee which will fulfil the reservation. It can only be done after the Order Item is created (visible in diagram <n>).

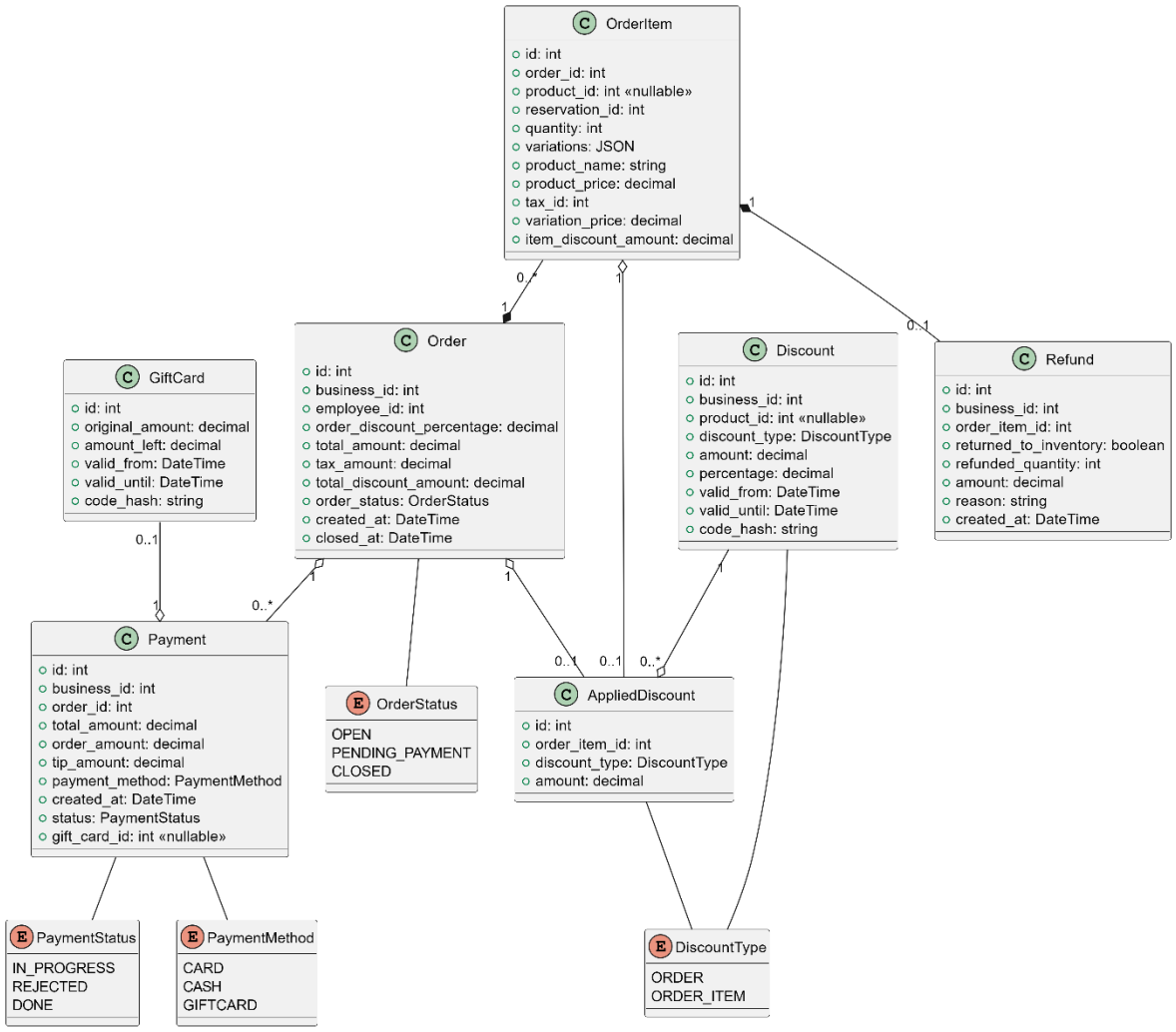

Figure 22: Data model of the order part of the business. Made with: www.plantuml.com

An Order is created by an employee and belongs to the business. It contains information about all the total amounts to make the system more efficient. Discounts can be applied to Order Items or the Order itself. The snapshots of Discounts are saved in the Applied Discount class to store the relation with Order or Order Item and the historical data of that discount.

An Order is made of Order Items. The total amount of the Order depends on the prices of the Order Items, their variations, taxes, discounts and quantities. When the Order is fully configured, a Payment process is initiated. Payment belongs to an Order. Payments of an Order can be split and paid via different payment methods, such as Cash, Credit or a Gift Card. Gift Cards have the original amount and the amount that is left after some money was withdrawn from it to a successful payment. The order is only closed when all the payments total amounts equal the total amount of the order.

This diagram highlights the relationship between an order item, reservation and a product (see Figure 23). Originally, the order item gets most of its information from the Product that has been assigned at the time of the creation. However, all of the data required for the historical records is kept in the Order Item class directly, therefore the ability to delete the Product or modify its fields is retained without compromising the integrity of the historical data. In the case of Products deletion, the product_id in the OrderItem becomes null. This does not apply to the relationship between OrderItem and Reservation. All reservations are kept in the historical records with their state tracked in the ReservationStatus enum. As a consequence, the service_id field of the Reservation becomes null as well when the Product gets deleted. This does not compromise the historical data because it is preserved in the OrderItem.

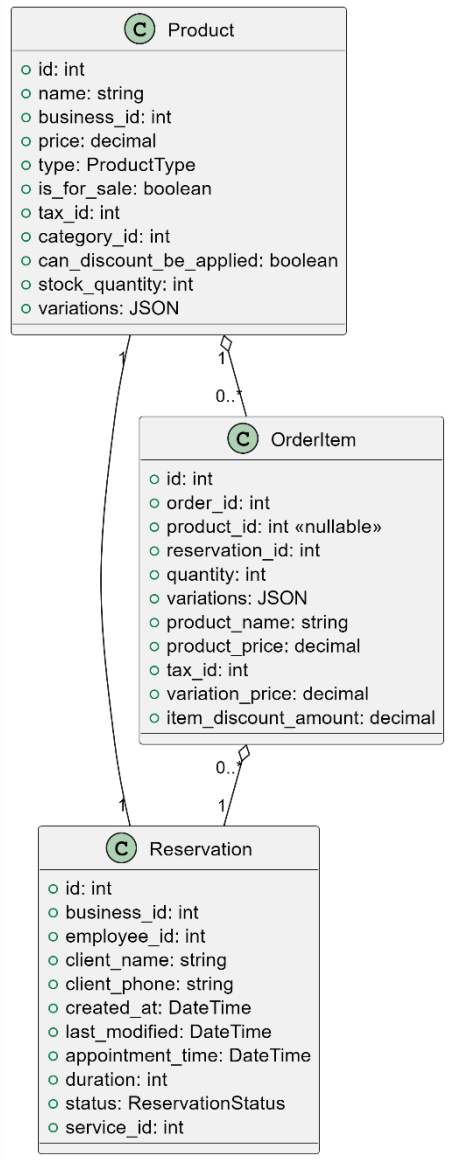

Figure 23: Data model concerned with reservations. Made with: www.plantuml.com

ENUMS

In this section, we define the key enumerators used throughout the system.

1. **Order Status**
    - **OPEN**
      The order has been created and the employee can add order items.
      Transitions:
        - Can change to **PENDING_PAYMENT.**
    - **PENDING_PAYMENT**
      Awaiting payments. No more items can be added once a payment is made.
      Transitions:
        - Can change to **CLOSED** once payment is completed.
    - **CLOSED**
      The order and all associated items are finalized and locked.
      Transitions:
        - Cannot be changed from this state.

2. **Payment Method**
    - **GIFTCARD**
      Payment made using a gift card.
    - **CASH**
      Payment made in cash.
    - **CARD**
      Payment made using a card.

3. **Payment Status**
    - **IN_PROGRESS**
      Payment is awaiting processing.
      Transition:
        - Can change to **DONE** after successful processing.
        - Can change to **REJECTED** if payment fails.
    - **REJECTED**
      Payment failed due to insufficient funds or other errors.
      DONE
      Payment has been successfully processed.

4. **Reservation Status**
    - **RESERVED**
      The item or service has been reserved.
      Transition:
        - Can change to **DONE** after the reservation is fulfilled.

    - Can change to **CANCELLED** if the reservation is voided.
- **CANCELLED**
  Reservation was cancelled.
- **DONE**
  Reservation has been completed.

5. **Employee Role**
   - **OWNER**
     Full administrative access with ownership rights.
     SUPER_ADMIN
     Extended administrative access, typically for IT support.
     Transitions:
   - **EMPLOYEE**
     General access for regular employees.
     Transitions:
     - Can change to **OWNER** through promotion.

6. **Product Type**
   - **SERVICE**
     A product classified as a service.
   - **ITEM**
     A tangible item.
   - **SERVICE_CHARGE**
     Additional charge for services.

7. **Account Status**
   - **ACTIVE**
     Employee is currently active and working.
     Transitions:
     - Can change to **INACTIVE** when on leave or temporarily unavailable.
     - Can change to **LEFT** if terminated.
   - **INACTIVE**
     Employee is temporary inactive (e.g., on vacation).
     Transitions:
     - Can change to **ACTIVE** upon return.
     - Can change to **LEFT** if no return.
   - **LEFT**
     Employee no longer works for the business.
8. **Discount Type**
   - **ORDER**
     Discount for the whole order.
   - **ORDER_ITEM**
     Discount for an order item.

Payment system integration

We chose *STRIPE* as the payment provider, because it is easier and quicker to integrate with.

Recommended video for reference: https://youtu.be/1r-F3FIONl8?si=6GLsQlogpq6OqNXS

In short, to integrate with Stripe:

1. You send the items, their quantity, and the price of each item (in cents) to Stripe.
2. Stripe returns you a URL which is the checkout page that contains the items you sent etc.
3. Then you can open the URL (checkout page), and stripe does the rest. Credit card details, person details, integrate with banks etc, basically the *actual* payment part.
4. Once checkout is completed (whether it was cancelled / rejected / processed successfully etc) -> Stripe will send back a response to your configured URLs.

So, the flow with payment integration will look something like this

1. User chooses *CARD* payment method and for what / how much to pay
2. A POST /payment request to our system is made to indicate that payment (with specified amount) is *IN_PROGRESS*.
3. Then *UI* sends the details to *STRIPE*
4. Once checkout is completed, *STRIPE* indicates whether payment was processed successfully / rejected etc.
5. Based on *STRIPE's* response, act accordingly. If payment was rejected, change payment status in our system to *REJECTED*, if payment was successful, change payment status in our system to *DONE*.

API Contract

The Swagger specification YAML file of the OMGVA PoS API is attached with the document (in the OMGVA_API.yaml file).

Error code reason explanation:

- 200 – operation completed successfully.
- 201 – object was created successfully.
- 204 – operation completed successfully and did not return anything.
- 400 – bad request, some fields are missing from the request body, are malformed or the body contains something that should not be there.
- 401 – endpoint is accessed without a valid authentication token.
- 403 – the user privilege level is insufficient to perform the operation
- 404 – the object that the request tries to access does not exist or could not be found.
- 405 – the object status does not allow the operation to be performed on it.
- 422 – the request data is semantically incorrect, e. g. multiple reservations are assigned to the same employee at the same time or more items are tried to be refunded than were bought.
- 5XX – a reminder that the server can fail unexpectedly and throw an error.