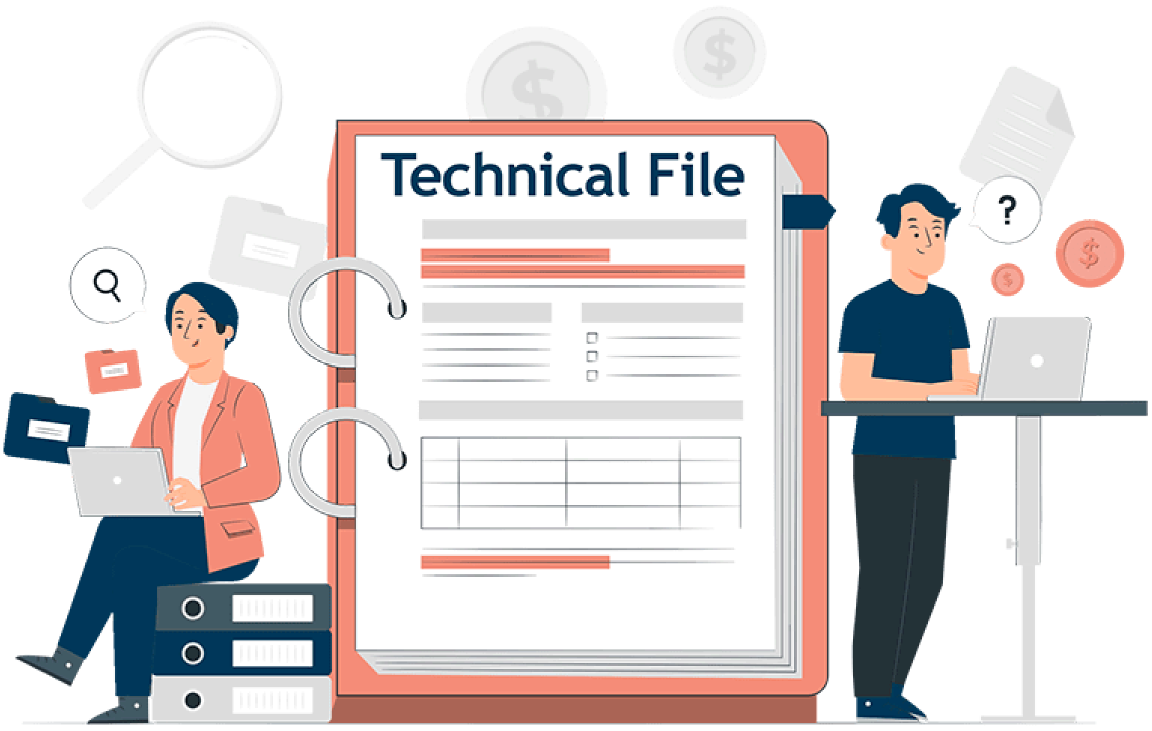
GLOBAL CODERS💻

Lab. assignment 1. *Software Design*



Alima Kassenkazy
Hayat Moawad
Alper Özgür Şahin
Carlos Alonso Rodríguez
Santiago Ramírez Olivares
Aleksandr Sventickij

Content

Intro

This document outlines the technical design of a **Software as a Service (*SaaS*)** system aimed at small and medium-sized businesses in the catering (*bars, cafés, restaurants*) and beauty (*barbershops, hairdressers, spas*) sectors. The system is designed for business employees to manage orders and reservations, optimize customer service processes, and streamline the administration of products, services, and payments.

The purpose of this system is to provide an all-in-one solution that allows businesses to efficiently handle their daily operations. Key features include the creation and modification of orders, payment processing (*with integration to payment providers such as Stripe or Elavon*), reservation management, as well as the management of inventory, taxes, discounts, and users.

The main objective of this project is to build a robust and scalable system that allows employees to perform tasks swiftly and accurately, while the system architecture is designed to support future expansions.

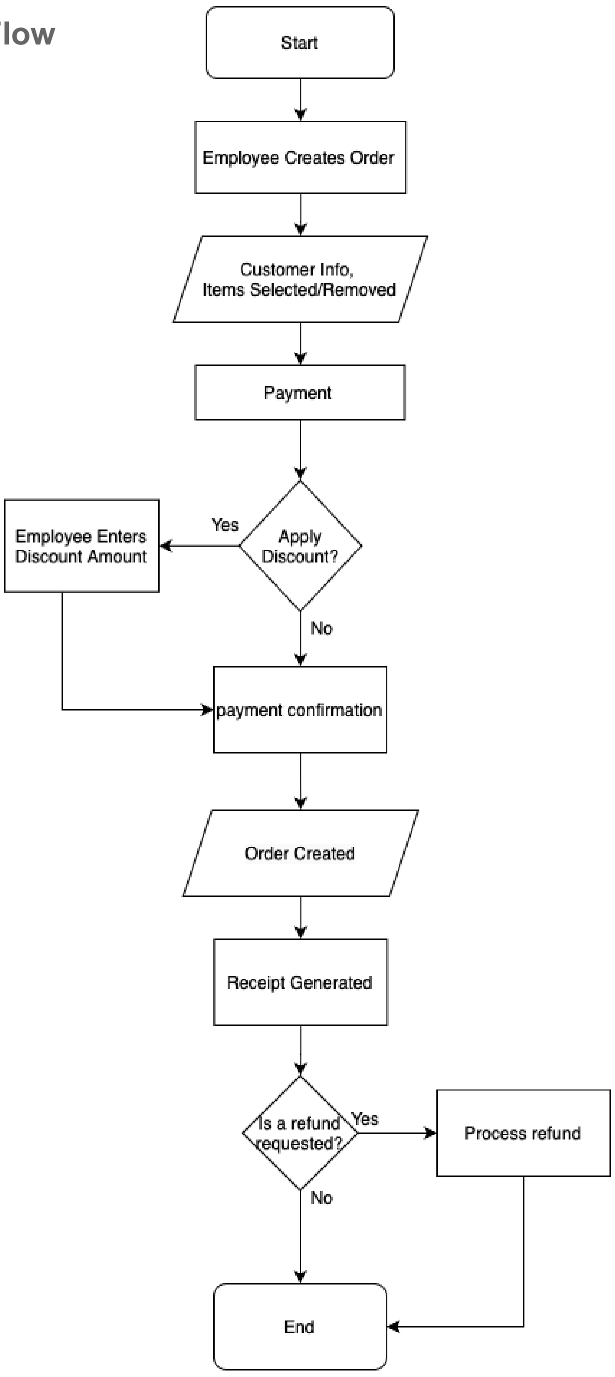
This document details the business flows, high-level architecture, data models, and API contracts necessary for developers to implement the system effectively. By the end of the project, users should be able to seamlessly manage orders, reservations, payments, and business administration, ensuring operational efficiency and customer satisfaction.

Business flows

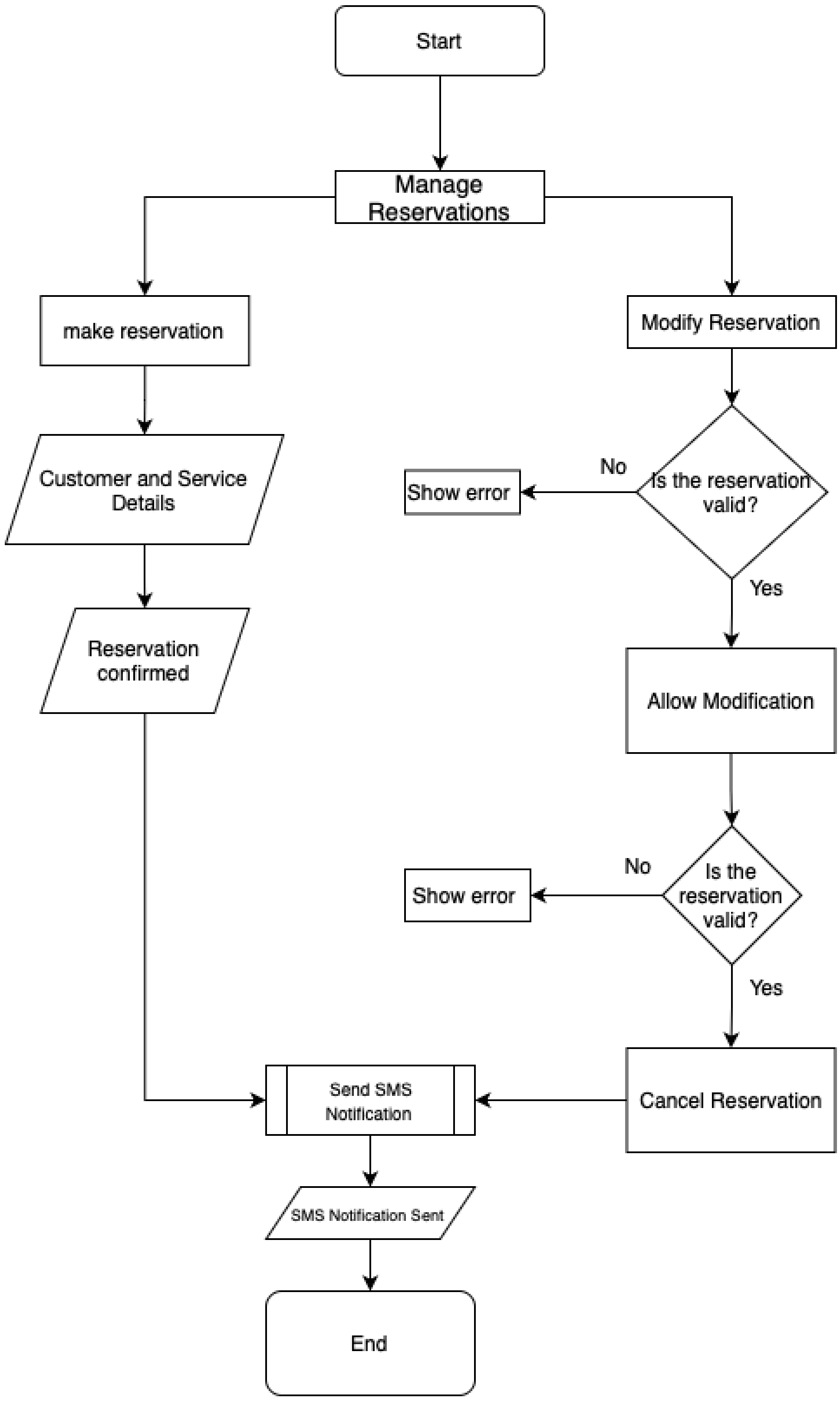
This section outlines the key business processes that the system supports, detailing how different users (*employees, customers*) interact with the system to manage daily operations.

The flows cover essential functionalities such as *order management, payment processing, reservation scheduling, and system administration*. Each process is described step-by-step, highlighting user actions and system responses, with visual aids such as sequence diagrams to clarify interactions.

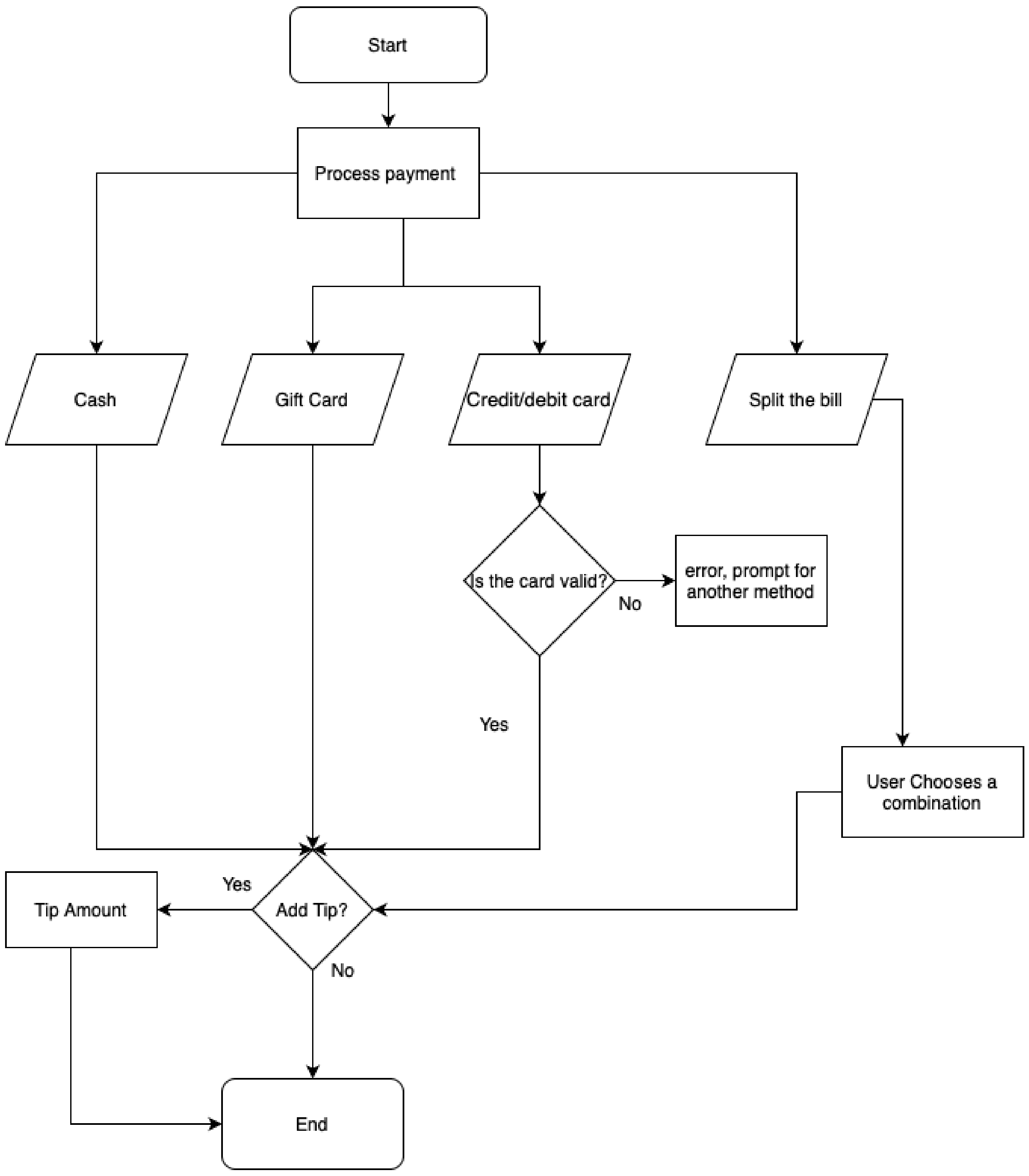
Order Management Flow



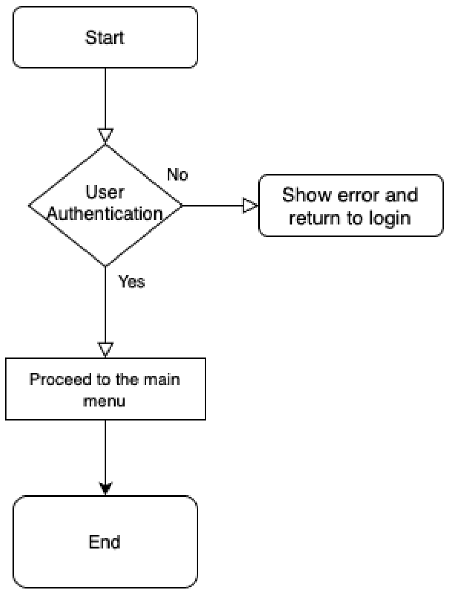
Reservation Scheduling Flow



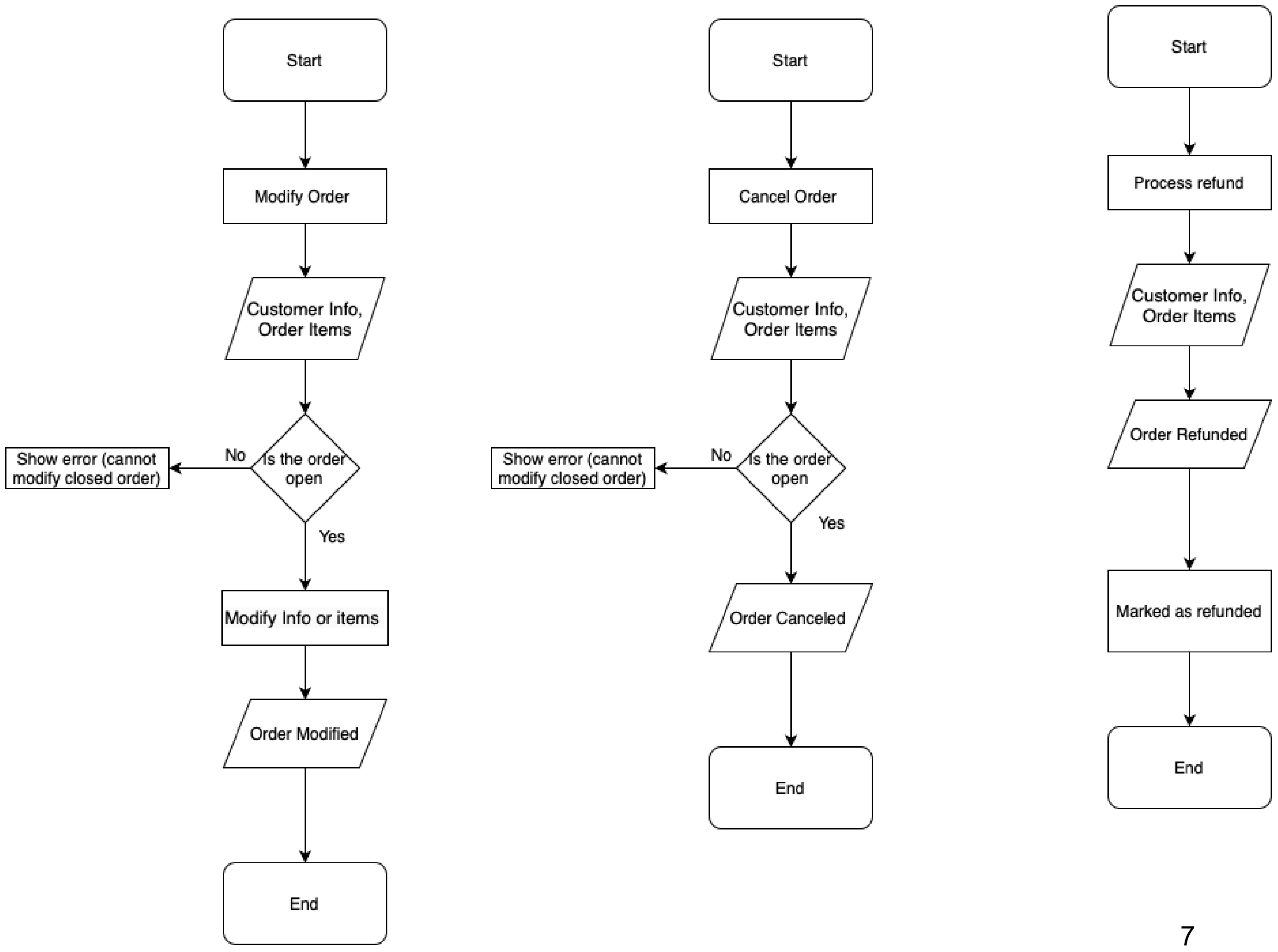
Payment Processing Flow



User Authentication Checking Flow



Modification/Cancellation/Refund Process Flow



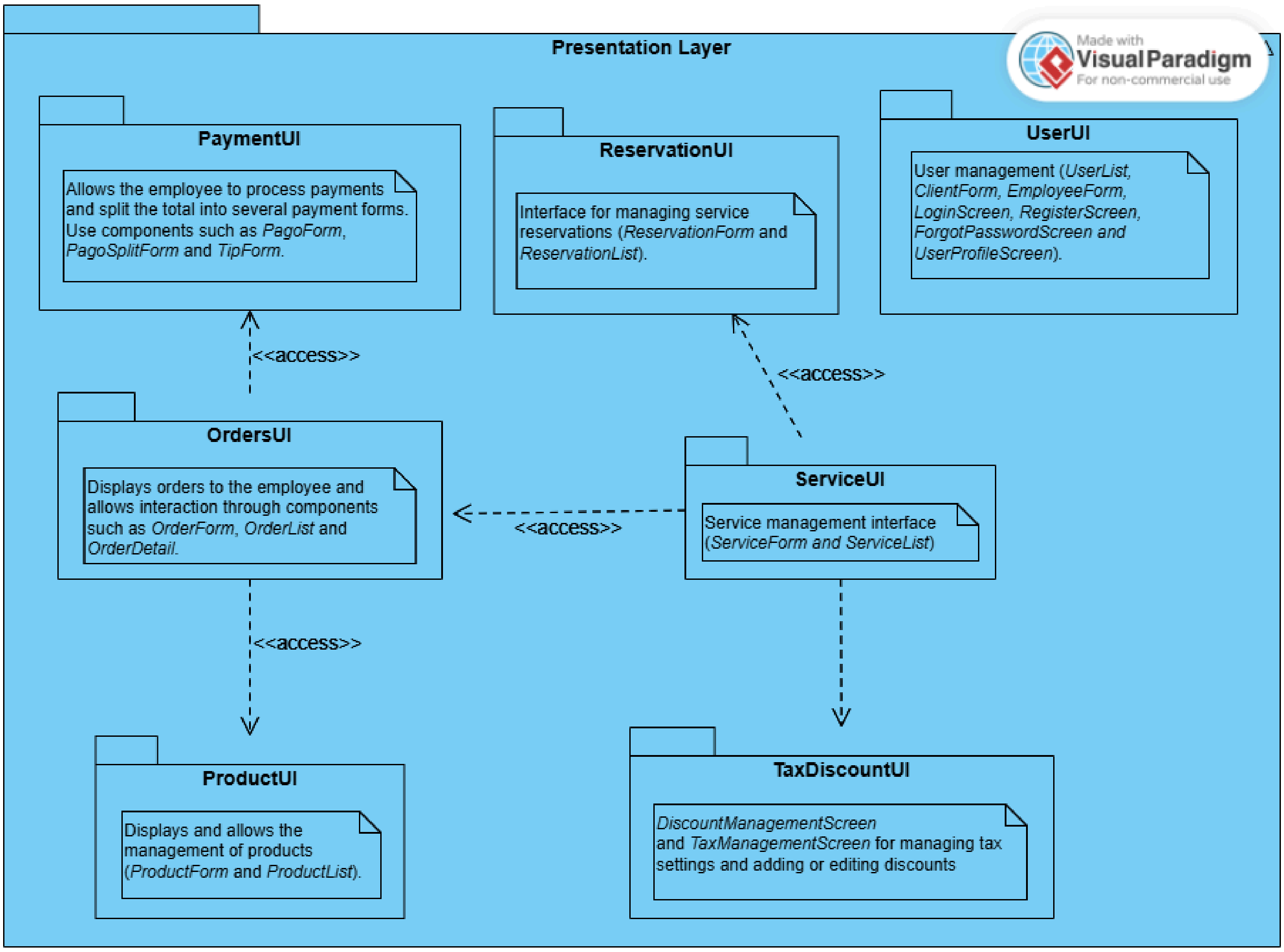
High level architecture

The architecture of the system is designed with a modular approach, comprising four primary layers: **Presentation Layer**, **Business Layer**, **Data Layer**, and **Service Layer**. Each layer has distinct responsibilities, facilitating a clear separation of concerns and enhancing maintainability and scalability.

Presentation Layer

This layer is responsible for user interaction and displays various user interfaces. It includes packages such as **OrdersUI**, **PaymentsUI**, **ReservationsUI**, **UsersUI, etc**. Each package provides specific functionalities, allowing employees to create and manage orders, process payments, make reservations, and handle user management.

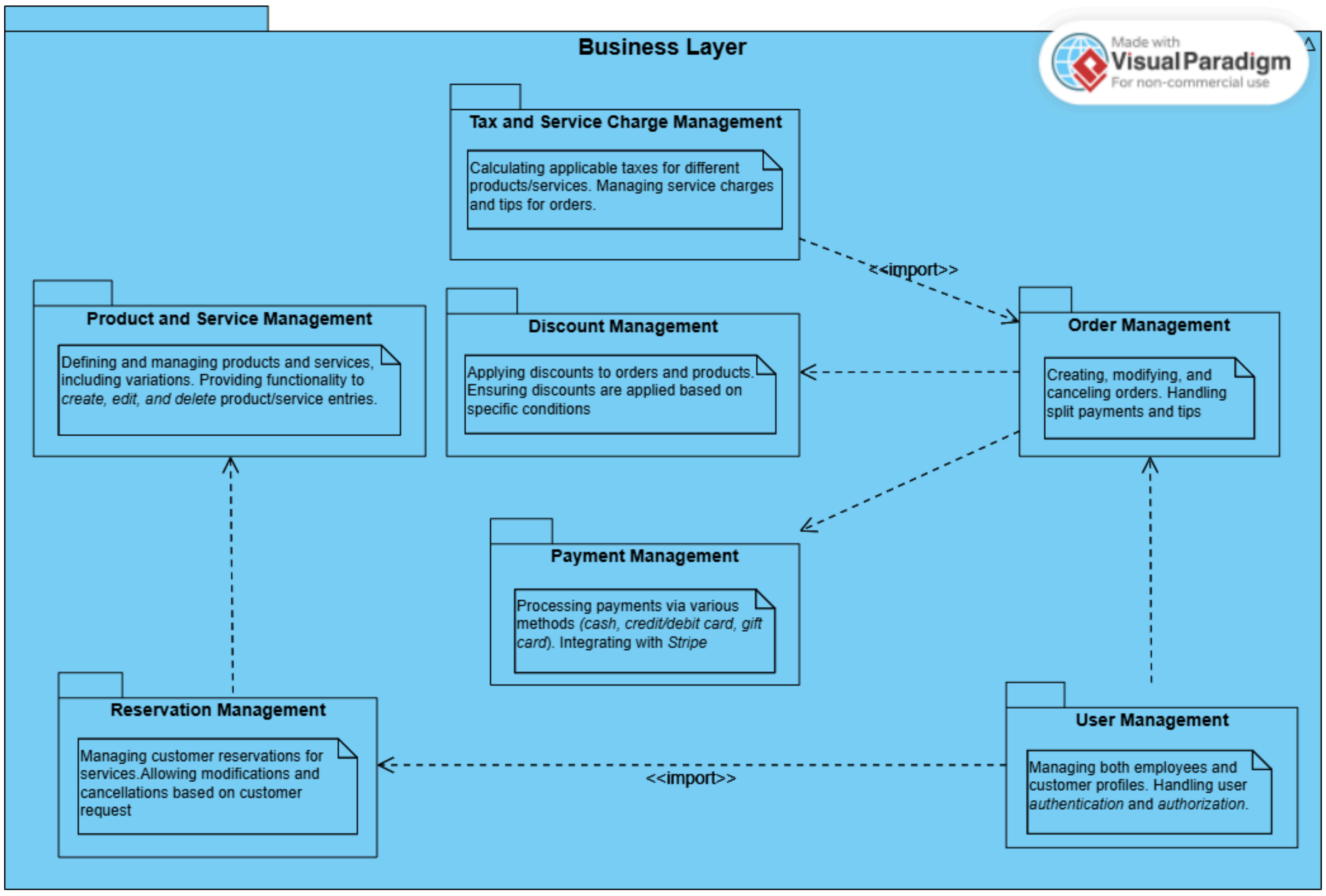
Example: An employee uses the **OrdersUI** to create a new order. This action triggers the business logic in the **Business Layer**.


Presentation Layer
Made with Visual Paradigm For non-commercial use
PaymentUI
Allows the employee to process payments and split the total into several payment forms. Use components such as PagoForm, PagoSplitForm and TipForm.
ReservationUI
Interface for managing service reservations (ReservationForm and ReservationList).
UserUI
User management (UserList, ClientForm, EmployeeForm, LoginScreen, RegisterScreen, ForgotPasswordScreen and UserProfileScreen).
<<access>>
<<access>>
OrdersUI
Displays orders to the employee and allows interaction through components such as OrderForm, OrderList and OrderDetail.
ServiceUI
Service management interface (ServiceForm and ServiceList)
<<access>>
<<access>>
ProductUI
Displays and allows the management of products (ProductForm and ProductList).
TaxDiscountUI
DiscountManagementScreen and TaxManagementScreen for managing tax settings and adding or editing discounts

Business Layer

The **Business Layer** contains the core business logic of the application, encapsulated in packages such as **OrderManagement**, **PaymentProcessing**, **ReservationManagement**, **UserManagement, etc**. This layer orchestrates the interactions between the presentation and data layers, ensuring that business rules are applied consistently.

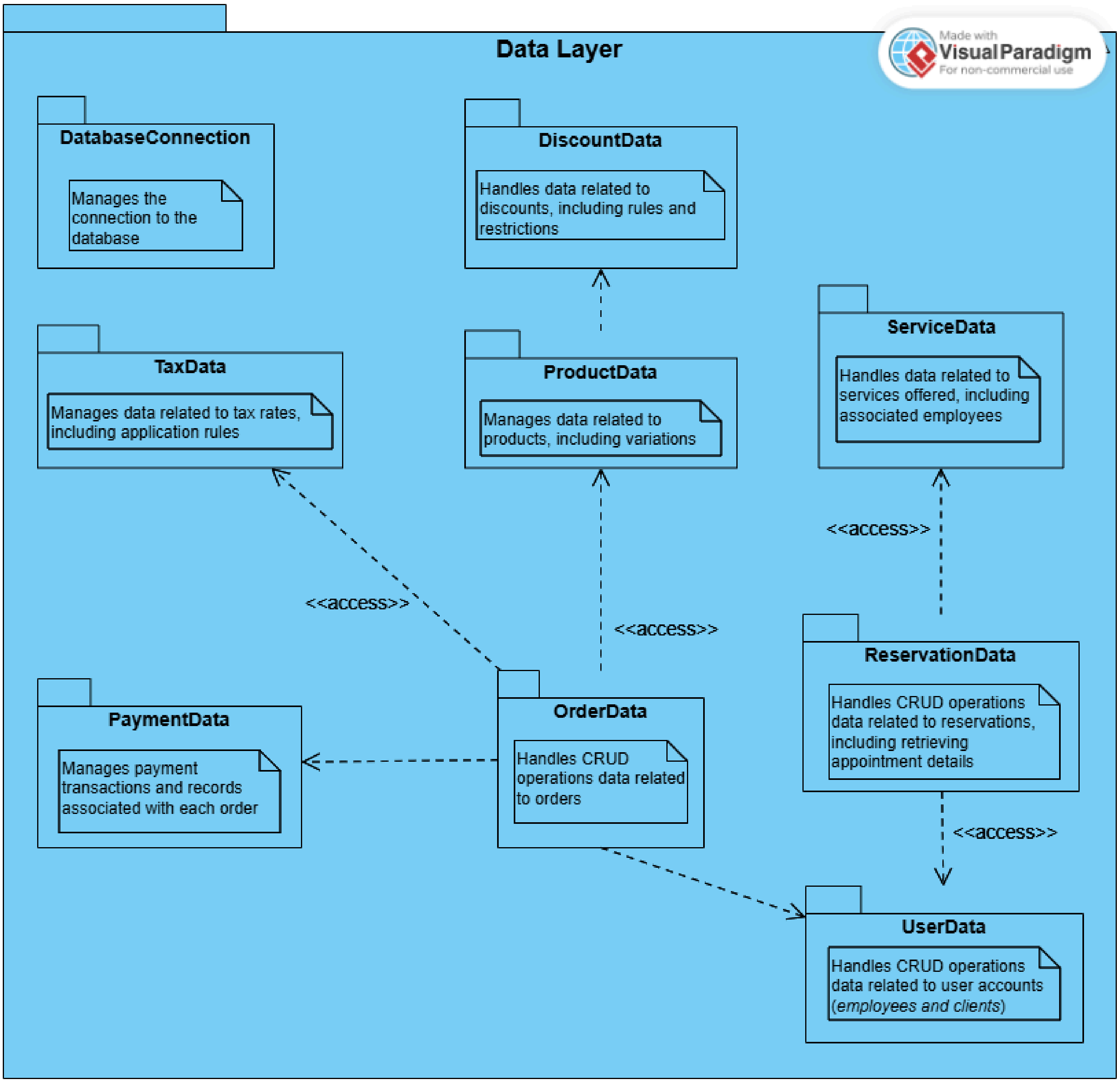
Example: When an order is created in the **Presentation Layer**, the **OrderManagement** package validates the order details, calculates taxes, and prepares the order for payment processing.


Business Layer
Made with Visual Paradigm For non-commercial use
Tax and Service Charge Management
Calculating applicable taxes for different products/services. Managing service charges and tips for orders.
<<import>>
Product and Service Management
Defining and managing products and services, including variations. Providing functionality to create, edit, and delete product/service entries.
Discount Management
Applying discounts to orders and products. Ensuring discounts are applied based on specific conditions
Order Management
Creating, modifying, and canceling orders. Handling split payments and tips
Payment Management
Processing payments via various methods (cash, credit/debit card, gift card). Integrating with Stripe
Reservation Management
Managing customer reservations for services.Allowing modifications and cancellations based on customer request
<<import>>
User Management
Managing both employees and customer profiles. Handling user authentication and authorization.

Data Layer

The **Data Layer** is responsible for data persistence and retrieval. It includes packages like **OrderData**, **UserData**, **ProductData**, and **ReservationData**. This layer interfaces directly with the database, handling CRUD operations for various entities.

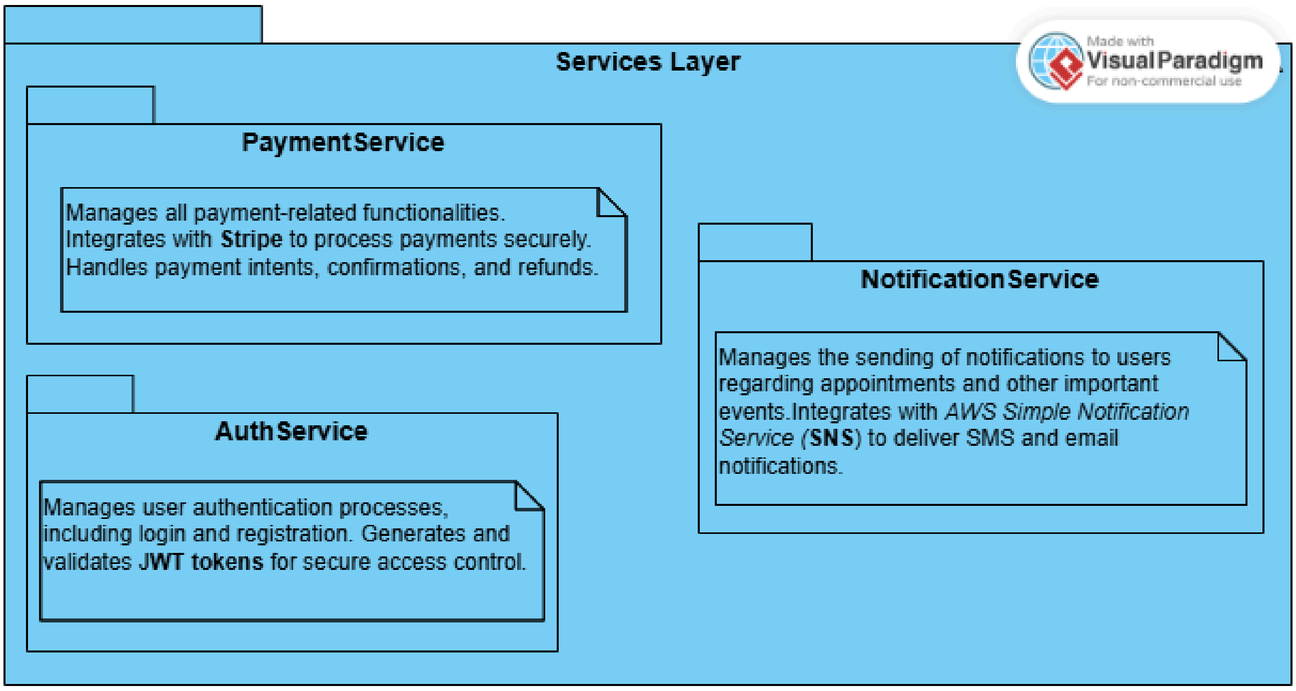
Example: After an order is created and validated in the **Business Layer**, the **OrderData** package saves the order details in the database, ensuring that it can be retrieved later for processing payments or generating reports.


Data Layer
Made with Visual Paradigm For non-commercial use
DatabaseConnection
Manages the connection to the database
DiscountData
Handles data related to discounts, including rules and restrictions
TaxData
Manages data related to tax rates, including application rules
ProductData
Manages data related to products, including variations
ServiceData
Handles data related to services offered, including associated employees
<<access>>
<<access>>
<<access>>
ReservationData
Handles CRUD operations data related to reservations, including retrieving appointment details
PaymentData
Manages payment transactions and records associated with each order
OrderData
Handles CRUD operations data related to orders
<<access>>
UserData
Handles CRUD operations data related to user accounts (employees and clients)

Service Layer

The **Service Layer** manages integrations with external services and security protocols. It consists of packages like **PaymentService**, **NotificationService**, and **AuthService**. This layer provides functionalities such as processing payments through Stripe, sending notifications via AWS SNS, and handling user authentication with JWT.

<u>*Example*</u>: Once an order is confirmed and payment details are provided, the **PaymentService** communicates with Stripe to create a payment intent and process the transaction securely.


Services Layer
PaymentService
Manages all payment-related functionalities. Integrates with Stripe to process payments securely. Handles payment intents, confirmations, and refunds.
NotificationService
Manages the sending of notifications to users regarding appointments and other important events.Integrates with AWS Simple Notification Service (SNS) to deliver SMS and email notifications.
AuthService
Manages user authentication processes, including login and registration. Generates and validates JWT tokens for secure access control.
Made with Visual Paradigm For non-commercial use

Inter-Layer Relationships

The layers interact seamlessly to deliver a cohesive user experience. For instance, when an employee creates an order in the **Presentation Layer**, the flow proceeds as follows:

1. The **Presentation Layer** sends the order details to the **Business Layer** for validation and processing.
2. The **Business Layer** retrieves necessary data from the **Data Layer** to store the order and calculate applicable taxes.
3. Upon successful order creation, the **Business Layer** calls the **Service Layer** to process payment and send notifications to the user.

This structured architecture ensures that each component of the system can evolve independently, while maintaining a clear path of interaction between layers. To understand the function of this layered architecture, let's look at an example:

Scenario: *Creating an Order*

<u>Step 1:</u> User Interaction - ***Presentation Layer***

1. **Employee Action** → An employee logs into the system through the **LoginUI** (part of the **Presentation Layer**) and navigates to the **OrdersUI** to create a new order.

2. **Input** → The employee fills in the order details (*e.g., selected items, quantities, customer information*) and submits the order.

<u>Step 2:</u> Order Submission - ***Business Layer***

3. **Request Handling** → The **OrdersUI** sends a request to the **OrderManagement** component in the **Business Layer** to create a new order.

4. **Validation** →
   - **OrderManagement** validates the input (*e.g., checking item availability, ensuring customer information is complete*).
   - It may also calculate applicable taxes or discounts at this stage based on predefined business rules.

Step 3: Data Persistence - ***Data Layer***

5. **Data Storage →**
    - After validation, the **OrderManagement** component calls the **OrderData** package in the **Data Layer** to store the new order.
    - The **OrderData** component saves the order details in the database, ensuring all necessary information is recorded.

Step 4: Payment Processing - ***Service Layer***

6. **Payment Intent Creation →**
    - Once the order is created, the **OrderManagement** component invokes the **PaymentService** in the **Service Layer** to process the payment.
    - The **PaymentService** creates a payment intent with Stripe, specifying the amount and order details.

Step 5: Confirmation and Notification - ***Service Layer***

7. **Payment Confirmation →**
    - The employee and customer are directed to the payment interface where they can choose a payment method (*e.g., credit card, cash, gift card*).
    - After the payment is completed, the **PaymentService** confirms the transaction with Stripe.

8. **Notification →**
    - Upon successful payment confirmation, the **OrderManagement** component calls the **NotificationService** to send a confirmation message (*e.g., via email or SMS*) to the customer.
    - The **NotificationService** retrieves the customer's contact information from the **UserData** package in the **Data Layer** and sends the notification.

System Components Overview

The system follows a **modular design** to ensure scalability, maintainability, and ease of future expansion. It is divided into several core components, each responsible for a specific area of functionality, with clear communication pathways between them.

The key components include the **frontend interface**, **backend server**, **database**, **authentication** and integrations with **external services** like payment gateways and SMS notifications.

1. User Interface (*Frontend*) - Angular

This component is the interface used by employees to manage orders, reservations, and business settings. The frontend communicates with the backend via a RESTful API. **Build it using Angular**.

2. Application Logic (*Backend*) - Spring Boot

The backend handles all business logic and data processing, including order and reservation management, payment processing, user authentication, and business administration. **It can be developed using Spring Boot**, a robust Java framework for building scalable and maintainable RESTful services.

3. Database - MySQL/PostgreSQL

The system uses a **relational database** (*MySQL/PostgreSQL*) to store information about orders, customers, products, services, and business settings. The database schema ensures data integrity and supports complex queries to optimize system performance.

4. Authentication - JWT

The system can use JWT (*JSON Web Tokens*) **for secure user authentication and authorization**. JWT provides a stateless, token-based mechanism for authenticating users, which is ideal for distributed systems like the one being designed. It ensures that only authorized users can access sensitive operations, such as modifying orders or managing user accounts.

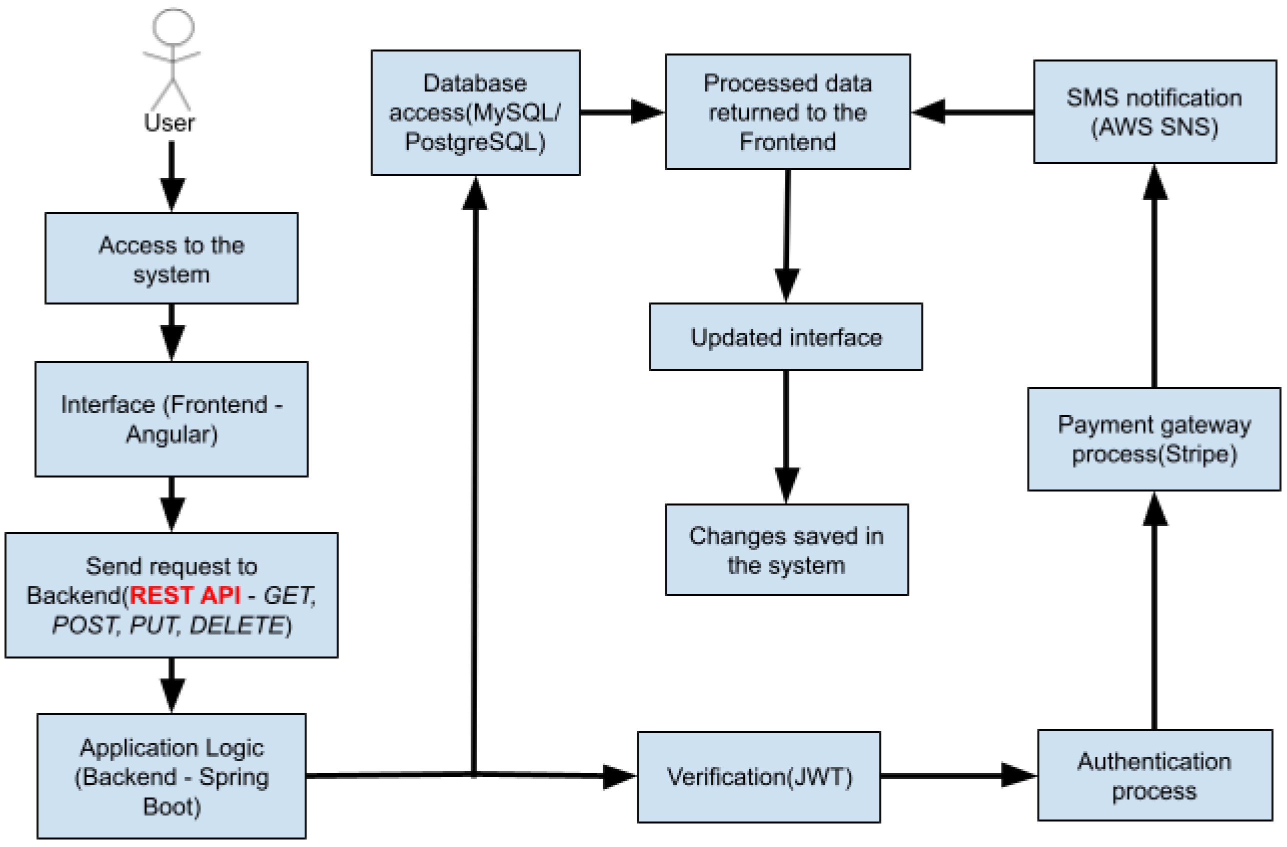
5. Payment Processing - Payment Gateway (e.g., Stripe)

This system integrates a Stripe payment gateway that leverages its API to handle payments securely and efficiently. Stripe offers a complete flow that includes payment processing, card handling, integration with alternative payment methods, etc. Once the user's payment information is securely captured, the transaction is processed and generates receipts, payment confirmations and other related items.

6. SMS Notification Service

This integration allows the system to send SMS notifications for important events, such as appointment confirmations, reminders, and cancellations. The backend will interface with a third-party SMS service provider, such as **AWS SNS, to send messages.** Ensure that customer phone numbers and any other sensitive information are stored securely and comply with relevant data protection regulations (*e.g., GDPR*).

The communication between the frontend and backend happens through an **API REST**, which uses standard HTTP methods (*GET, POST, PUT, DELETE*) to interact with data.


User
Access to the system
Interface (Frontend - Angular)
Send request to Backend(REST API - GET, POST, PUT, DELETE)
Application Logic (Backend - Spring Boot)
Database access(MySQL/ PostgreSQL)
Processed data returned to the Frontend
Updated interface
Changes saved in the system
Verification(JWT)
Authentication process
Payment gateway process(Stripe)
SMS notification (AWS SNS)

Class diagram

One of the most important parts of software design is to **identify the entities involved and how they relate to meet the system requirements**. In this section, a class diagram is shown where you can find different parts, each class with its respective attributes, methods and relationships with the rest of the components. For this we have used different colors that represent **different groupings**. Below is a brief legend with its corresponding explanation:

- **Users**

  The classes involved in this group are: *Customer, Owner, Employee and SuperAdmin.* A business must have an **owner**, an **IT administrator** who can manage the infrastructure (*in this case, edit any user*), a number of **employees** to be able to work and finally, **customers** who consume the services and products.

- **Order**

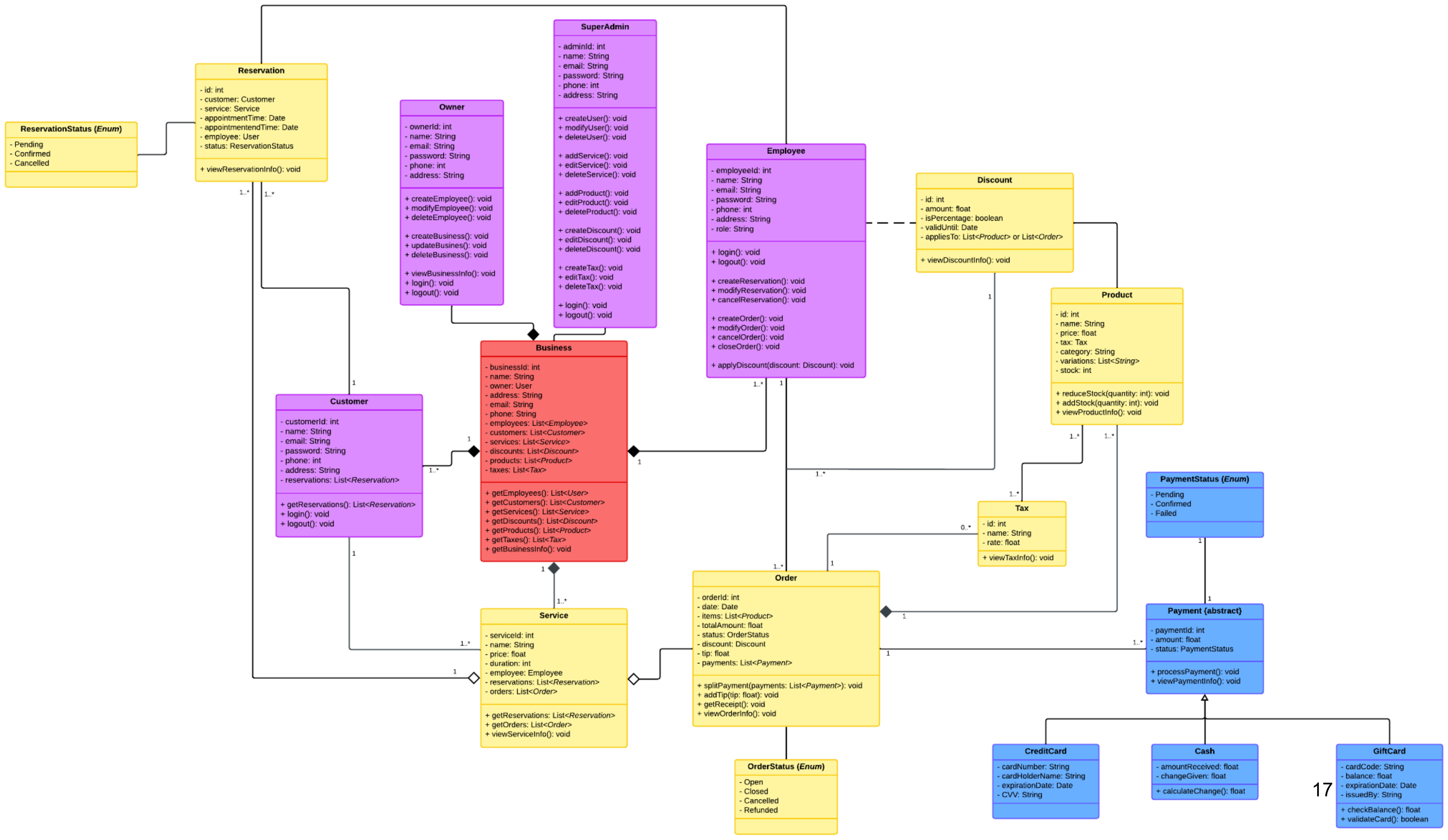
  In this group we can find the *Order, Discount, Product, Service, Reservation* and *Tax* classes, as well as the enumerations *OrderStatus* and *ReservationStatus*. In any business related to the catering or beauty sector, a **service** must be offered. Each service must have an **order** that will sell **products** to which certain **taxes** are applied. Likewise, a customer may want a service but at another time, so they will be able to make **reservations** through employees. In addition, there may be offers on some products or even employees can apply **discounts** on orders.

- **Payment**

  A business needs money, so it's important to set up a good **payment** system. To do this, customers will be able to pay with a **credit or debit card**, a **gift card**, or simply **cash**.

- **Business**

  The main class of the diagram, the business. Through this entity we find the information of the entire company, from the name of the owner to its employees, customers and its services.


ReservationStatus (*Enum*)
- Pending
- Confirmed
- Cancelled
Reservation
- id: int
- customer: Customer
- service: Service
- appointmentTime: Date
- appointmentendTime: Date
- employee: User
- status: ReservationStatus
+ viewReservationInfo(): void
Owner
- ownerId: int
- name: String
- email: String
- password: String
- phone: int
- address: String
+ createEmployee(): void
+ modifyEmployee(): void
+ deleteEmployee(): void
+ createBusiness(): void
+ updateBusines(): void
+ deleteBusiness(): void
+ viewBusinessInfo(): void
+ login(): void
+ logout(): void
SuperAdmin
- adminId: int
- name: String
- email: String
- password: String
- phone: int
- address: String
+ createUser(): void
+ modifyUser(): void
+ deleteUser(): void
+ addService(): void
+ editService(): void
+ deleteService(): void
+ addProduct(): void
+ editProduct(): void
+ deleteProduct(): void
+ createDiscount(): void
+ editDiscount(): void
+ deleteDiscount(): void
+ createTax(): void
+ editTax(): void
+ deleteTax(): void
+ login(): void
+ logout(): void
Employee
- employeeId: int
- name: String
- email: String
- password: String
- phone: int
- address: String
- role: String
+ login(): void
+ logout(): void
+ createReservation(): void
+ modifyReservation(): void
+ cancelReservation(): void
+ createOrder(): void
+ modifyOrder(): void
+ cancelOrder(): void
+ closeOrder(): void
+ applyDiscount(discount: Discount): void
Discount
- id: int
- amount: float
- isPercentage: boolean
- validUntil: Date
- appliesTo: List<*Product*> or List<*Order*>
+ viewDiscountInfo(): void
Product
- id: int
- name: String
- price: float
- tax: Tax
- category: String
- variations: List<*String*>
- stock: int
+ reduceStock(quantity: int): void
+ addStock(quantity: int): void
+ viewProductInfo(): void
Customer
- customerId: int
- name: String
- email: String
- password: String
- phone: int
- address: String
- reservations: List<*Reservation*>
+ getReservations(): List<*Reservation*>
+ login(): void
+ logout(): void
Business
- businessId: int
- name: String
- owner: User
- address: String
- email: String
- phone: String
- employees: List<*Employee*>
- customers: List<*Customer*>
- services: List<*Service*>
- discounts: List<*Discount*>
- products: List<*Product*>
- taxes: List<*Tax*>
+ getEmployees(): List<*User*>
+ getCustomers(): List<*Customer*>
+ getServices(): List<*Service*>
+ getDiscounts(): List<*Discount*>
+ getProducts(): List<*Product*>
+ getTaxes(): List<*Tax*>
+ getBusinessInfo(): void
PaymentStatus (*Enum*)
- Pending
- Confirmed
- Failed
Tax
- id: int
- name: String
- rate: float
+ viewTaxInfo(): void
Service
- serviceId: int
- name: String
- price: float
- duration: int
- employee: Employee
- reservations: List<*Reservation*>
- orders: List<*Order*>
+ getReservations: List<*Reservation*>
+ getOrders: List<*Order*>
+ viewServiceInfo(): void
Order
- orderId: int
- date: Date
- items: List<*Product*>
- totalAmount: float
- status: OrderStatus
- discount: Discount
- tip: float
- payments: List<*Payment*>
+ splitPayment(payments: List<*Payment*>): void
+ addTip(tip: float): void
+ getReceipt(): void
+ viewOrderInfo(): void
Payment {abstract}
- paymentId: int
- amount: float
- status: PaymentStatus
+ processPayment(): void
+ viewPaymentInfo(): void
OrderStatus (*Enum*)
- Open
- Closed
- Cancelled
- Refunded
CreditCard
- cardNumber: String
- cardHolderName: String
- expirationDate: Date
- CVV: String
Cash
- amountReceived: float
- changeGiven: float
+ calculateChange(): float
GiftCard
- cardCode: String
- balance: float
- expirationDate: Date
- issuedBy: String
+ checkBalance(): float
+ validateCard(): boolean

API contract

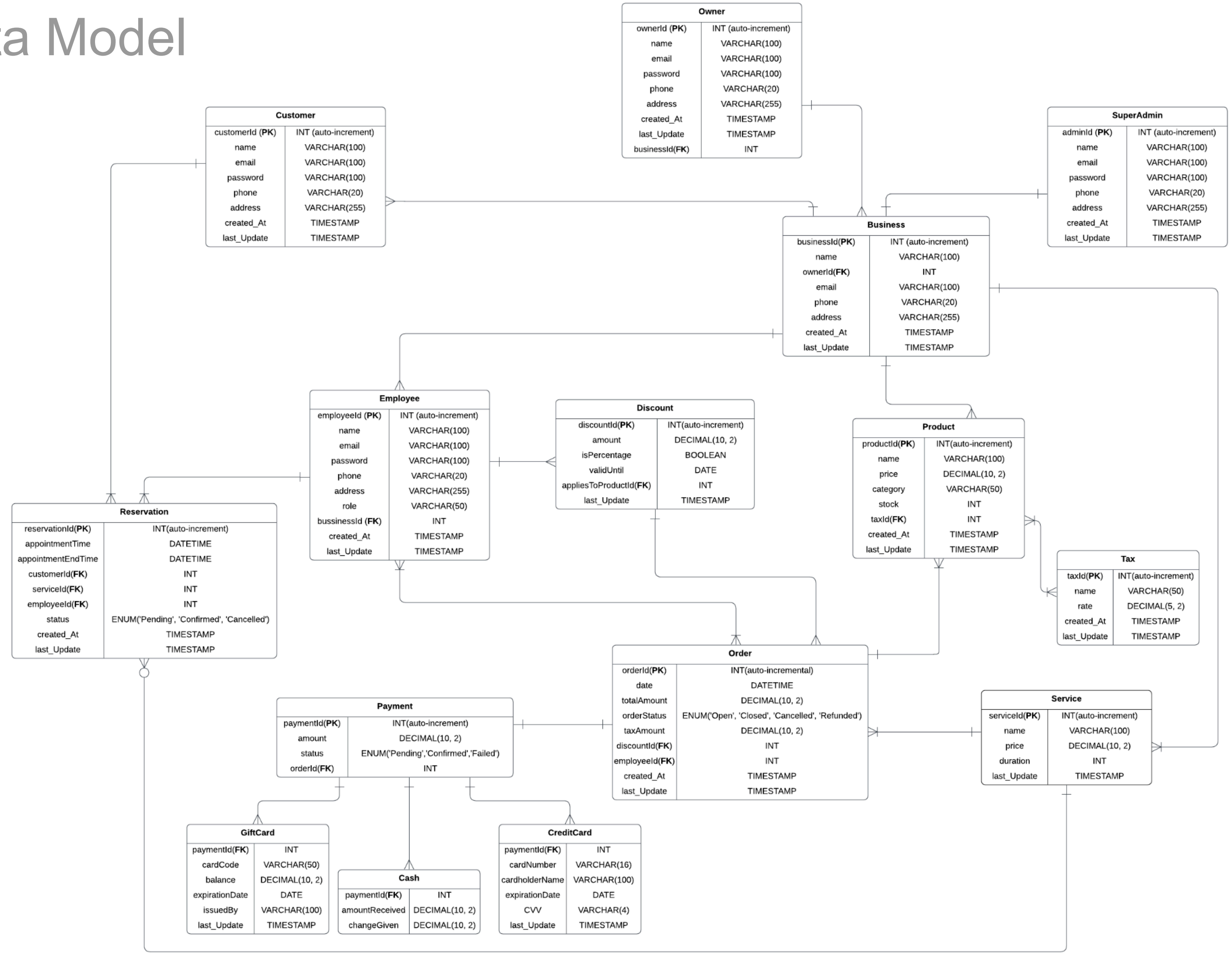
An API contract is the documentation that describes how the API works and how it should be used. When making changes to an API those changes should be communicated out and documented to avoid unexpected behavior from clients that consume it.

The API specifications for the project are defined in the "*GlobalCoders_API.yaml*" file. This file follows the **OpenAPI 3.0.3** standard and provides detailed information about the available endpoints, request/response formats, and security requirements.

To view and interact with the API specifications, you can use tools like Swagger Editor. **Swagger editor url:** https://editor.swagger.io/

Link to the Github with the yaml file → Sellita/Global_Coders_API (github.com)

Data Model



Owner	
ownerId (**PK**)	INT (auto-increment)
name	VARCHAR(100)
email	VARCHAR(100)
password	VARCHAR(100)
phone	VARCHAR(20)
address	VARCHAR(255)
created_At	TIMESTAMP
last_Update	TIMESTAMP
businessId(**FK**)	INT

Customer	
customerId (**PK**)	INT (auto-increment)
name	VARCHAR(100)
email	VARCHAR(100)
password	VARCHAR(100)
phone	VARCHAR(20)
address	VARCHAR(255)
created_At	TIMESTAMP
last_Update	TIMESTAMP

SuperAdmin	
adminId (**PK**)	INT (auto-increment)
name	VARCHAR(100)
email	VARCHAR(100)
password	VARCHAR(100)
phone	VARCHAR(20)
address	VARCHAR(255)
created_At	TIMESTAMP
last_Update	TIMESTAMP

Business	
businessId(**PK**)	INT (auto-increment)
name	VARCHAR(100)
ownerId(**FK**)	INT
email	VARCHAR(100)
phone	VARCHAR(20)
address	VARCHAR(255)
created_At	TIMESTAMP
last_Update	TIMESTAMP

Employee	
employeeId (**PK**)	INT (auto-increment)
name	VARCHAR(100)
email	VARCHAR(100)
password	VARCHAR(100)
phone	VARCHAR(20)
address	VARCHAR(255)
role	VARCHAR(50)
bussinessId (**FK**)	INT
created_At	TIMESTAMP
last_Update	TIMESTAMP

Discount	
discountId(**PK**)	INT(auto-increment)
amount	DECIMAL(10, 2)
isPercentage	BOOLEAN
validUntil	DATE
appliesToProductId(**FK**)	INT
last_Update	TIMESTAMP

Product	
productId(**PK**)	INT(auto-increment)
name	VARCHAR(100)
price	DECIMAL(10, 2)
category	VARCHAR(50)
stock	INT
taxId(**FK**)	INT
created_At	TIMESTAMP
last_Update	TIMESTAMP

Reservation	
reservationId(**PK**)	INT(auto-increment)
appointmentTime	DATETIME
appointmentEndTime	DATETIME
customerId(**FK**)	INT
serviceId(**FK**)	INT
employeeId(**FK**)	INT
status	ENUM('Pending', 'Confirmed', 'Cancelled')
created_At	TIMESTAMP
last_Update	TIMESTAMP

Tax	
taxId(**PK**)	INT(auto-increment)
name	VARCHAR(50)
rate	DECIMAL(5, 2)
created_At	TIMESTAMP
last_Update	TIMESTAMP

Order	
orderId(**PK**)	INT(auto-incremental)
date	DATETIME
totalAmount	DECIMAL(10, 2)
orderStatus	ENUM('Open', 'Closed', 'Cancelled', 'Refunded')
taxAmount	DECIMAL(10, 2)
discountId(**FK**)	INT
employeeId(**FK**)	INT
created_At	TIMESTAMP
last_Update	TIMESTAMP

Payment	
paymentId(**PK**)	INT(auto-increment)
amount	DECIMAL(10, 2)
status	ENUM('Pending','Confirmed','Failed')
orderId(**FK**)	INT

Service	
serviceId(**PK**)	INT(auto-increment)
name	VARCHAR(100)
price	DECIMAL(10, 2)
duration	INT
last_Update	TIMESTAMP

GiftCard	
paymentId(**FK**)	INT
cardCode	VARCHAR(50)
balance	DECIMAL(10, 2)
expirationDate	DATE
issuedBy	VARCHAR(100)
last_Update	TIMESTAMP

Cash	
paymentId(**FK**)	INT
amountReceived	DECIMAL(10, 2)
changeGiven	DECIMAL(10, 2)

CreditCard	
paymentId(**FK**)	INT
cardNumber	VARCHAR(16)
cardholderName	VARCHAR(100)
expirationDate	DATE
CVV	VARCHAR(4)
last_Update	TIMESTAMP

The ER diagram represents the key entities involved in managing a business system, including *customers*, *employees*, *businesses*, *orders*, *payments*, and more. The relationships between entities are defined using foreign keys (**FK**), ensuring the system maintains data integrity and consistency. The main entities include:

- **Customer**: Stores customer details like name, contact info, and registration data.
- **Employee**: Manages orders, reservations, and other business functions.
- **Order**: Central entity that links products/services, customers, employees, and payments.
- **Payment**: Tracks how customers pay for their orders using multiple methods (*credit card, cash, etc.*).
- **Product/Service**: Defines items or services that can be purchased and taxed.
- **Tax**: Stores applicable tax rates for products.
- **Discount**: Manages discounts applicable to orders or specific products.

Each entity contains necessary fields to store business data, like **timestamps to track record creation and updates**, ensuring system auditing and tracking.

This ER diagram is designed to capture the essential data relationships and structures needed to operate a business system, specifically for catering and beauty service sectors. It aligns with key data modeling principles by:

1. **Normalization**: The data model reduces redundancy by splitting data into separate, logically connected tables (*e.g.,* *`Customer, Employee, Order, Product, Payment`*). This ensures that each piece of data is stored only once, enhancing efficiency.
2. **Relational Integrity**: The diagram uses foreign key relationships (*e.g.,* `customerId, employeeId, orderId`) to connect tables, enforcing data consistency across related entities. This ensures, for example, that orders are always linked to a valid customer and employee.
3. **Scalability**: By separating entities like `Tax, Discount, Product,` and `Payment`, the model can easily accommodate new business rules or additional services, without requiring major structural changes.

4. **Data Integrity and Auditing**: Each table includes timestamps (`created_At`, `last_Update`) to track when records are created or modified. This allows for accurate auditing and ensures historical data integrity, especially important in business-critical applications.

5. **Flexibility**: The model allows for various payment methods (*e.g.,* `CreditCard`, `Cash`, `GiftCard`) and supports features like order refunds, service reservations, and business management (*e.g., through the* `Owner` *and* `SuperAdmin` *entities*). This ensures the system meets both business and operational needs while allowing easy data access and modification.

To design such diagrams, the simplest and most effective way would be to understand the use cases we face as users. Therefore, an example of how we can use this data design for a real use case is shown below.

Scenario: *Creating an Order*

1. **Customer places an order** for a product (from the `Product` table).
2. The **Order** is created, linking the customer (`customerId`), the employee managing it (`employeeId`), and the products/services (`productId`).
3. **Tax and discounts** are applied from the `Tax` and `Discount` tables to calculate the final amount.
4. The **Payment** is recorded, and the customer pays using a method stored in the `Payment` table *(e.g., credit card or cash*).
5. The order is updated with a **status** (`open`, `closed`), and a receipt is generated for the customer.

This interaction shows how entities are linked through relationships to ensure a seamless order process.