Point of sale system

Technical documentation

Introduction

The Point of Sale (POS) system is designed to streamline business operations by enabling employees to create, modify, and manage customer orders. It supports multiple payment methods, including cash, gift cards, and card payments (with future integration for Stripe or Elavon). Features include split payments, tip and discount application, and comprehensive tax management for different items.

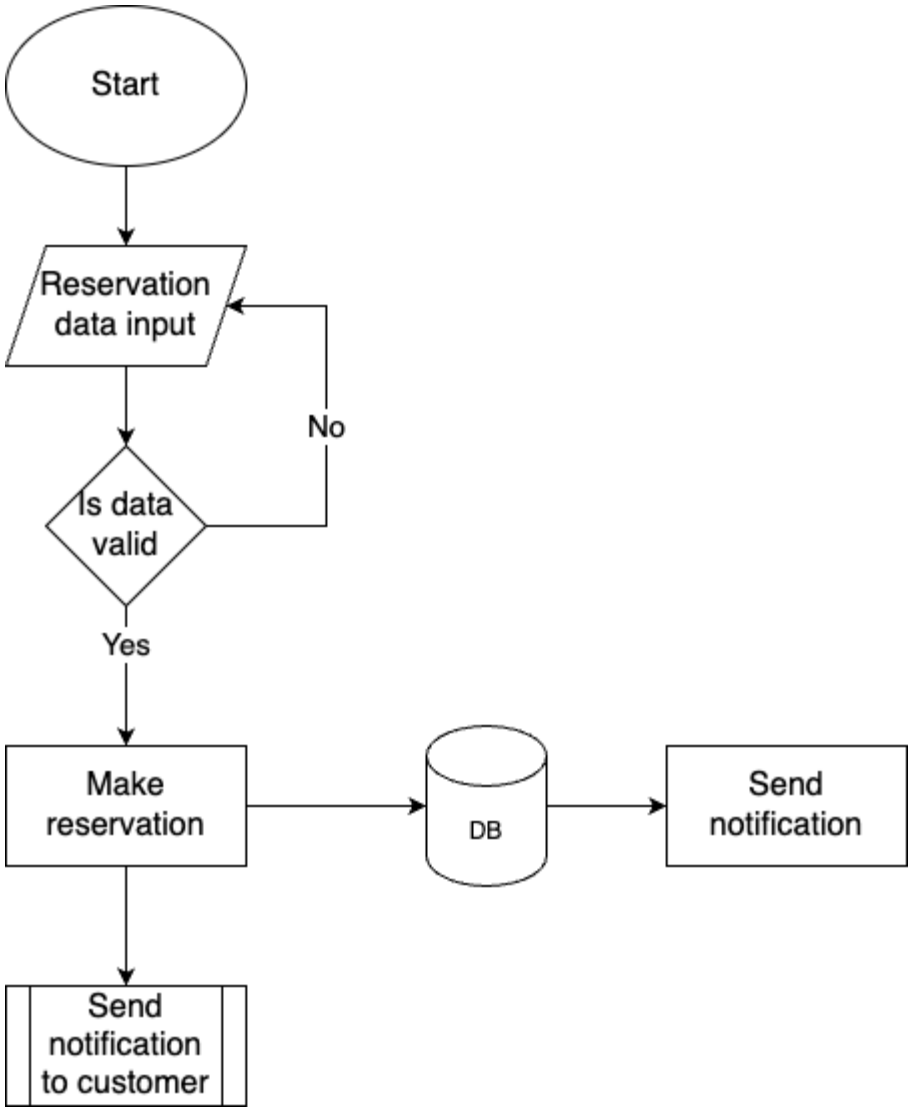
The system also offers inventory tracking, ensuring that stock is updated in real time. A robust user management system allows business owners and managers to control employee access and roles. Closed orders are preserved indefinitely, with options for refunds when necessary, ensuring efficient handling of transactions and customer service.

Flows

The main program flows are presented here. The blueish color is representing an inner process that will be shown in more detail in later flows.

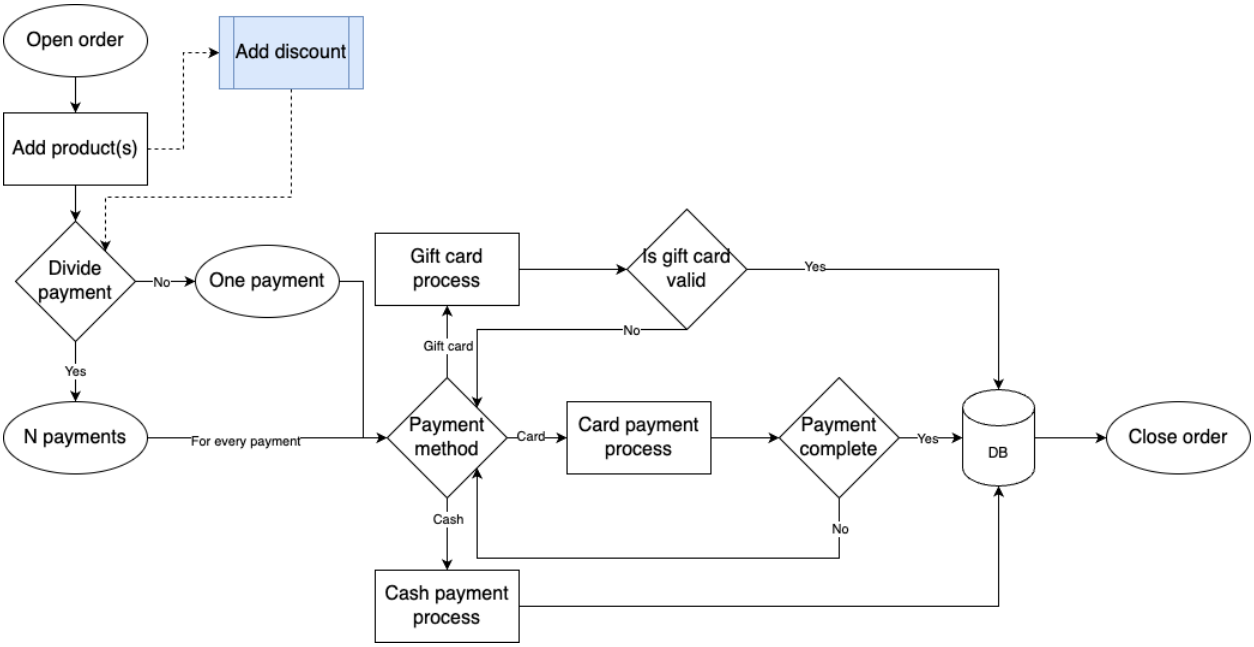
Reservation

Reservation could be made by the customer or employee. They are also able to cancel it. The reservation’s state would be changed.



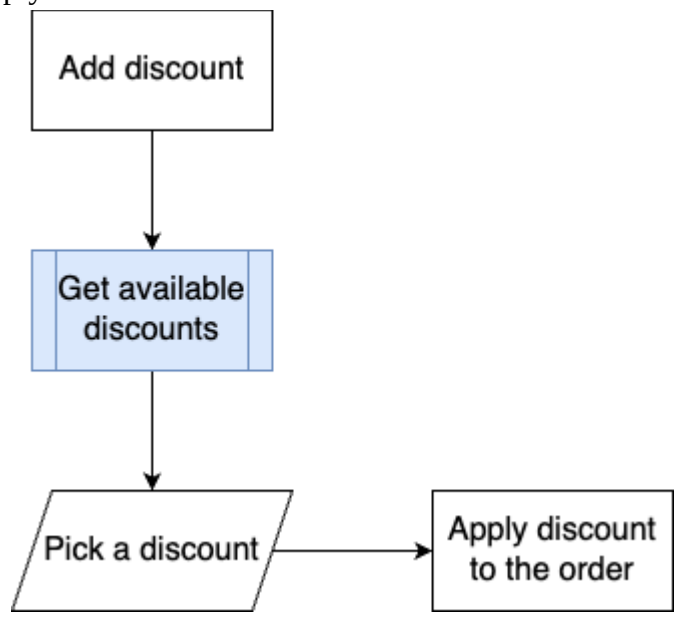
Order

Order flow is showing how order is being processed. Adding a discount is an optional process that may be started by an employee.



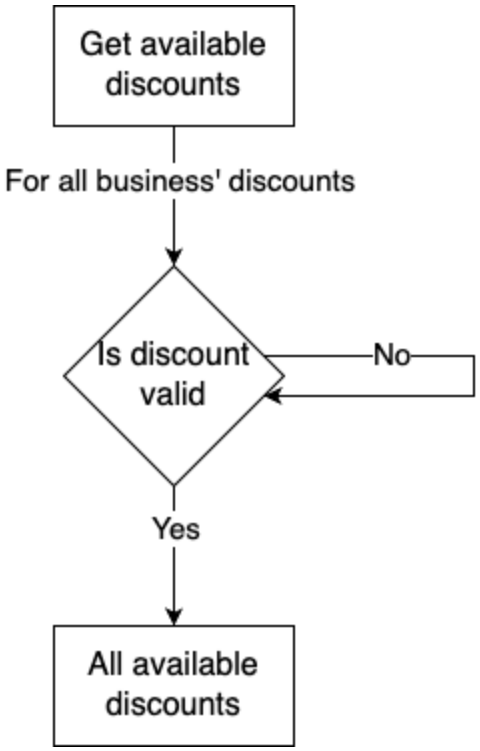
Order discount

Employees will be shown all available discounts on that order at the current time. Then the employee is able to apply the discount on the current order.



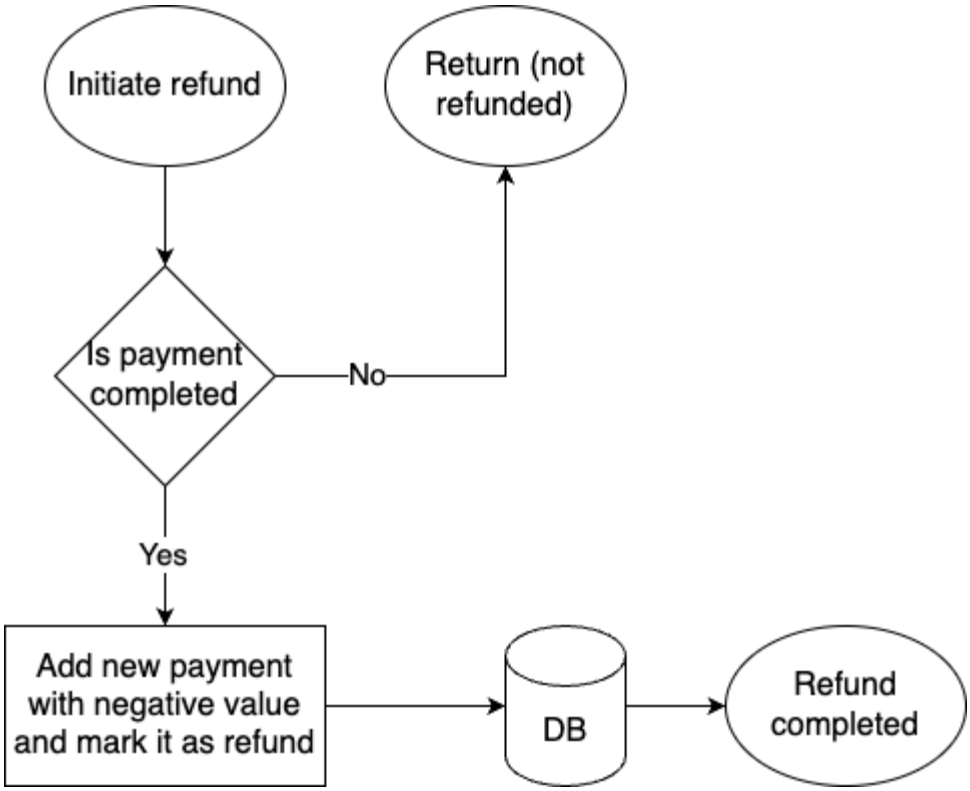
Available discounts

It is getting all available discounts at the current time. It is possible that some discounts are not valid at the current time of the day. Or they are not valid at all. Those will be skipped.



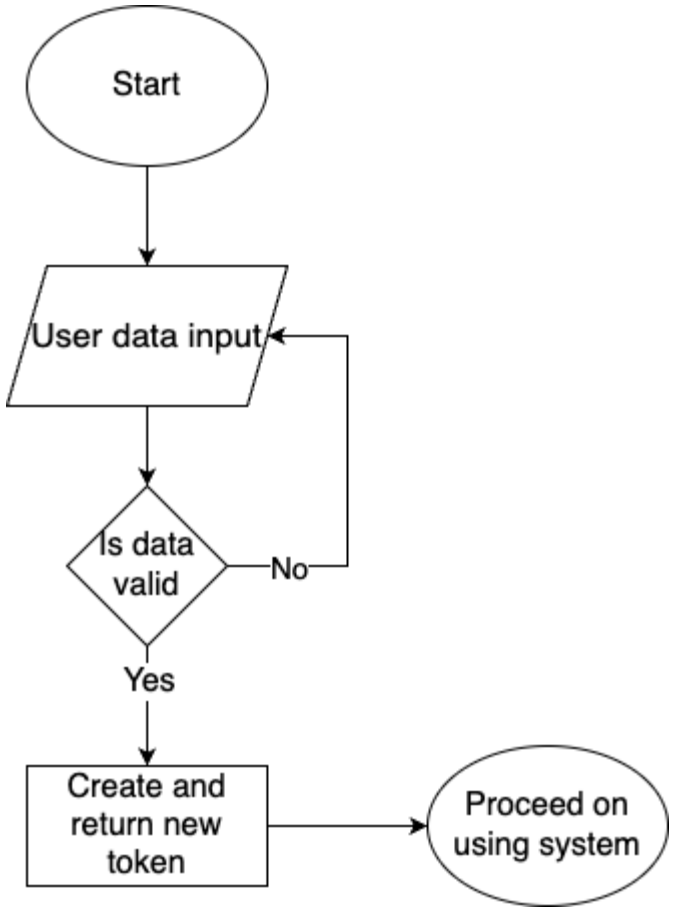
Refund

Refunds will be done this way. New entry of payment will be added with “Refunded” status enum and negative value of *PaymentSize* column.



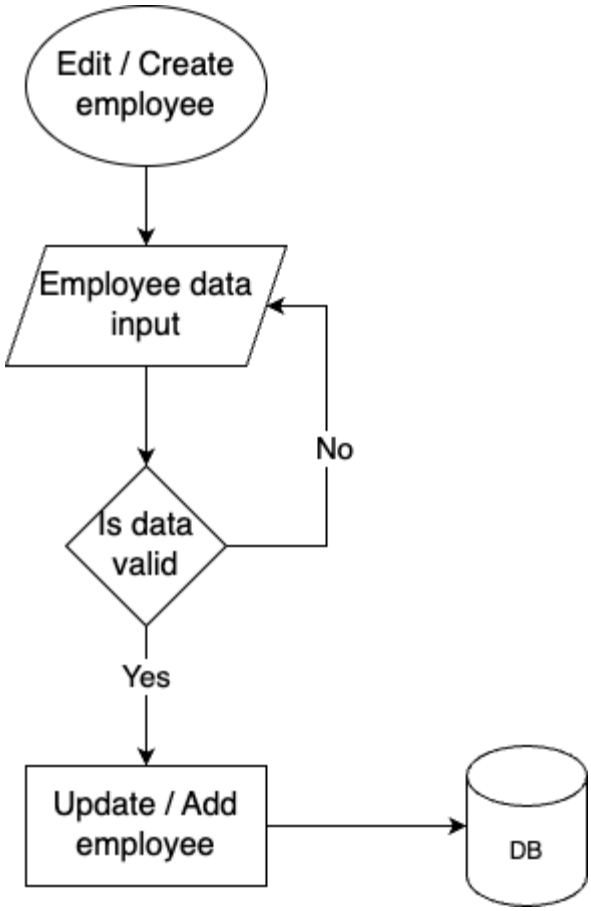
Log in

On login every user with the right credentials will get a token and be assigned his role. Then the user is able to proceed on using the point of sale system with the given token.



Employee

We are showing one create operation for clarity purposes. This flow creates an employee.



Data model

EventTime

Event Time is designed for applying discounts, and potentially for handling taxes in the future. It allows you to make discounts time-sensitive. For example, a discount could be valid from October 1, 2024 to November 1, 2024, between 12:00 and 15:00 every day until the end date.

Event Time stores the time in the **HH:MM** format, making it possible to check and compare the current time to determine if the discount applies.

System parameters

The `SystemParameters` table in a system can act as a centralized repository for global or system-wide (for every business) configuration settings that control how different aspects of the POS system behave. This table usually stores parameters that influence various modules or

functionalities in the system, allowing them to be dynamically updated without needing to modify code.

Examples:

1. **Notifications and Alerts:**
    a. **Email Server Configuration**: Stores SMTP server settings
    b. **SMS Gateway Settings**: Similar to email settings but for sending SMS notifications to customers or employees.
2. **Transaction Behavior and Limits:**
    a. **Maximum Discount Allowed**: A parameter that controls the maximum percentage or amount of discount a cashier or store manager can apply to any transaction.
    b. **Maximum Refund Period**: Defines the number of days a customer has to return items for a refund.

ModifiedBy and CreatedBy

Arrows should connect each entity with `CreatedBy` and `ModifiedBy` fields to the `User` table. However, we chose not to include them to avoid cluttering the diagram and making it harder to read.

Taxes

Each product can have multiple taxes assigned to it, allowing for flexibility in applying different tax rates or types, such as VAT, sales tax, or environmental fees, depending on the product and its regulatory requirements.

Product variation

Each product can have variations, allowing for different options such as size, color, or material. The `Product` table stores the base product information, while the `ProductVariation` table holds details about the specific variations that can be assigned to the product during an order.

Many to many relationships

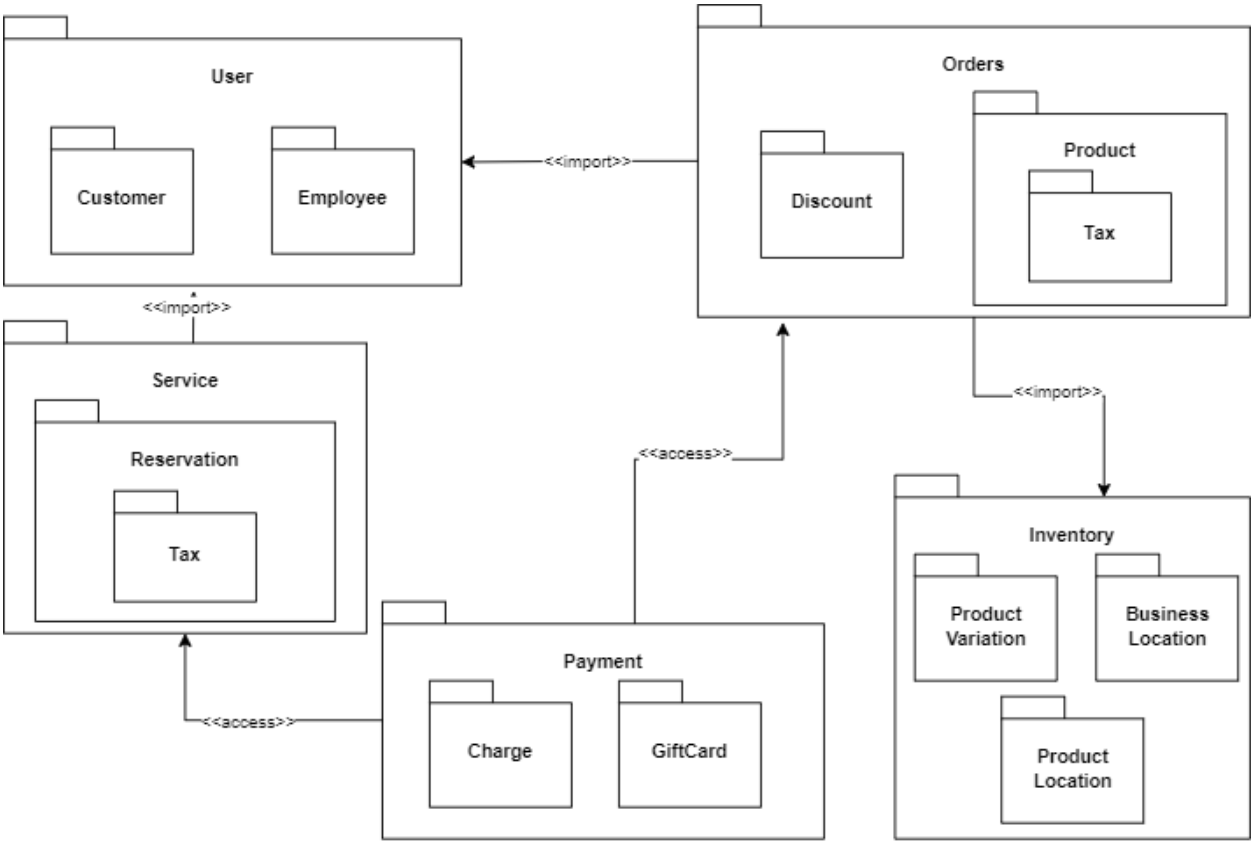
In a many-to-many relationship, each connection between two tables is represented by a new table. This junction table typically follows the naming convention: `Table1Table2Row`, combining the names of the related tables. The junction table contains references to both tables, allowing multiple records in one table to be associated with multiple records in the other.

Decimal data type

In this system, the decimal data type is used to store monetary values, ensuring precision and accuracy for financial data such as prices, taxes, and totals. This prevents rounding errors and guarantees reliable calculations in transactions and other money-related processes.

Package diagram

This package diagram shows the key parts of a point of sale system and how they interact. It includes **Customer**, **Employee**, **Orders**, and **Product**, which are essential for managing business operations. Other components like **Tax**, **Payment**, **Discount**, and **GiftCard** handle financial details and promotions. Inventory tracks stock levels, while **Product Location** and **Business Location** manage where products and business locations are stored. **Service** and **Reservation** deal with bookings and services offered. The **<<import>>** and **<<access>>** dependencies show how different parts of the system connect and rely on each other.



Roles and permissions

Roles

Each endpoint in our system is assigned a unique permission. For instance, permissions like:

- Employee.Get
- Employee.Put
- Employee.Post

and others will represent distinct permissions. These permissions are stored in the SystemModule table, where each is associated with a unique integer ID.

Additionally, businesses will have the flexibility to define custom roles, referred to as `UserTypes`, tailored to their specific needs. Each role can be assigned a set of permissions, which will be maintained in a many-to-many relationship table called `UserSystemModuleRow`.

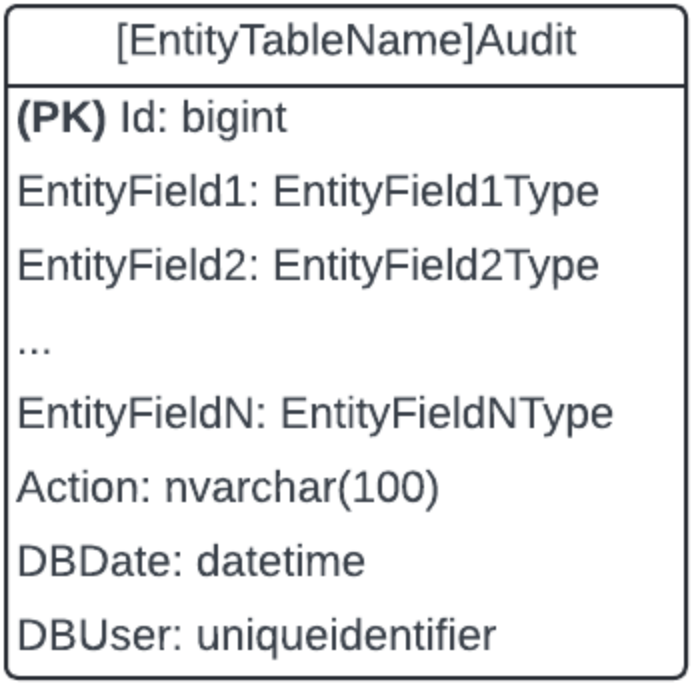
Data manipulation

Each endpoint should verify whether the user initiating the request intends to modify data belonging to their own business. If this condition is not met, an error message should be displayed to the user.

This could be done by middleware. In this approach, middleware acts as a centralized access control mechanism that ensures each user action in the system adheres to the appropriate permission boundaries. It operates by intercepting requests before they reach the actual application logic and evaluates whether the user initiating the request possesses the necessary permissions to proceed.

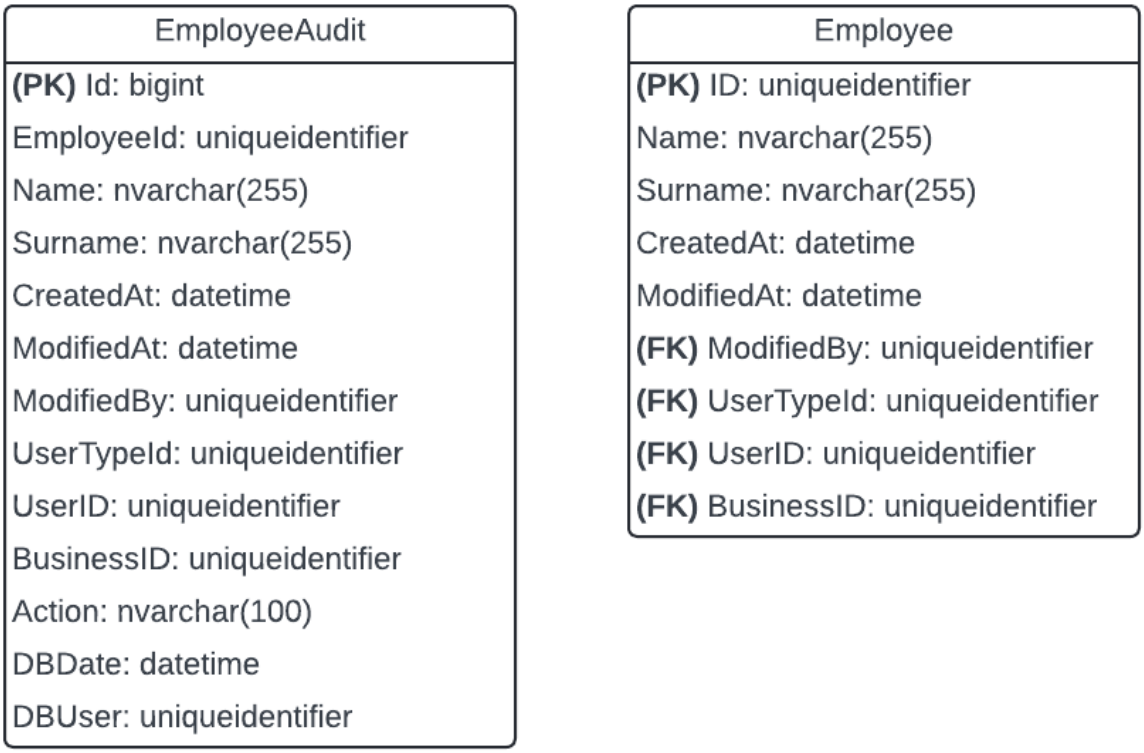
Audit data

A data audit should be implemented using triggers on every entity table, triggered by the actions `INSERT`, `UPDATE`, and `DELETE`.



The table presented here is an abstract audit table, containing every field from the corresponding entity table. In addition, it includes four extra fields: **Id**, **Action**, **DBDate**, and **DBUser**. Each entity table will have three triggers (one for each action). These triggers will insert the previous data into the audit table and populate the additional fields as follows:

- **Id**: An auto-incrementing integer.
- **Action**: A value from the set {`“Insert”, “Update”, “Delete”`}, representing the action performed on the entity table.
- **DBDate**: The date the action was performed.
- **DBUser**: The user who performed the action.



Audit table example. In summary, the audit table contains all the fields of the entity table, with the addition of these four new fields: **Id**, **Action**, **DBDate**, and **DBUser**.

API contracts

Everything related to the API contracts and this project could be found here: GitHub

Appendix

Audit variation A

We also considered an "Insert-only" data model, where data would not be deleted from the entity table but instead marked with a timestamp. The latest data would be retrieved based on this timestamp.
However, this approach did not seem ideal for several reasons:

- The main table would grow rapidly, which would slow down data retrieval.
- Our data model includes several many-to-many relationships, which would make data retrieval more complex in this scenario.

- Retrieving historical data from the audit table is faster and more efficient than doing so from the main entity table.