Pit Boss Casino Database

Final Problem Statement

Grant Wyness, Aidan Kirk, and Canon Maranda

CSSE333 Winter 2022-23

Contents

Executive Summary

This final report document will provide the initial problem that merited this type of project, the specific solution that we utilized for the problem, the major challenges we faced in creating the solution, as well as the overall design of the database used in said solution. There will also be an analysis of our solution's strengths and weaknesses, an appendix containing the final entity-relation diagram for our database, and a relational schema. Lastly, you will find a glossary, an external references list, and an index.

Introduction

This document is the final report on the Pit Boss project worked on by the Pit Bosses, a team consisting of Aidan Kirk, Canon Maranda, and Grant Wyness. Its purpose is to take the initial problem and look at the effectiveness of the solution implemented by the Pit Bosses. This analysis of the effectiveness includes feedback from the Pit Bosses on what they felt were the strengths and weaknesses of this solution and it will also revisit several key points made in the Security Analysis and Final Problem Statement.

Problem Description

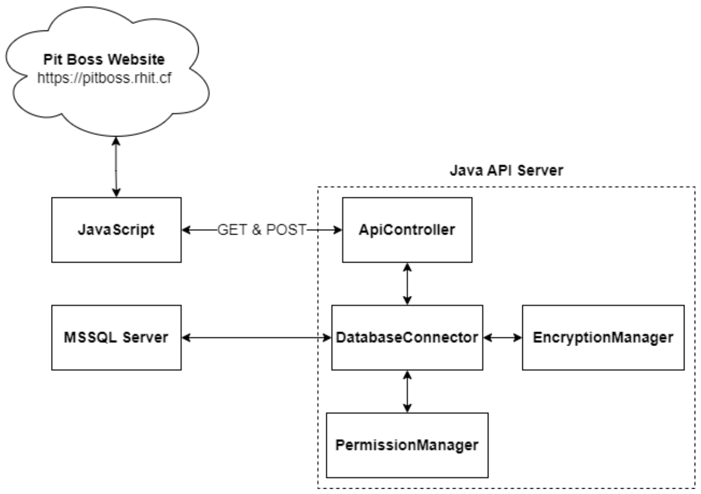
Casinos will often find themselves with a large amount of data on every single client, which can get very messy very quickly. Often, casinos will produce makeshift solutions to solve having so much data strewn around, such as Excel sheets or a very small, cheap database that does not actually track information vital to keeping the casino running. Often, vital things such as payment information for employees, current customer balances, locations of game tables, you name it, are missing from these smaller databases and excel spreadsheets. As such, there was a good opportunity present to establish a gold standard for casino database systems.

As such, the Pit Bosses set out to create this “gold standard.” The goal was to have an easy-to-use frontend for both prospective customers and casino employees that provides many of the essential features mentioned above, and a backend to hold onto all of it. This combination may be given to a prospective investor or casino data manager to provide a better alternative to current data storage measures.

Solution Description

Our solution utilizes a website-based application where customers and employees can access data. This data is stored in a Microsoft SQL database and accessible through a Java API, written with Spring Boot. Application users will authenticate with the Pit Boss website using their Driver’s License (DL) number, which is to be scanned into the system. Data requests are determined by the JavaScript frontend code, which sends either a GET or POST request to the API server. The API then sends a request to the database server through a stored procedure or table-valued function. Once the data is retrieved, it is sent up the request chain to the frontend user.

Data security plays an important role in our solution. We expect the database administrator and those who review logs to be reasonable people, such that they do not have ill intentions with user data. Any sensitive information that is sent to the frontend (driver's license numbers) is encrypted against a key that is inaccessible outside of the API server. The API server is also protected against secure user credentials and permission management.



Frontend Discussion

The Pit Boss website was designed with the Bootstrap 5 framework to offer a simple and modern design. Each page offers unique functionality that is relevant to a customer, employee, manager, or administrator. When visiting the login page, the user is required to input their Driver's License number. In practice, this will be entered using a barcode scanner. If the user does not exist, they are taken to a registration page to enter any required customer information. Otherwise, the login is successful and the user is directed to a starting page based on their role. An encrypted user identifier is stored in the browser's session storage as the authentication token.

Customers have access to a profile page, where they can view their personal information. They can deposit and withdraw from their casino account, view their 10 most recently played games, and update their name if necessary. This page is exclusive to customers and is not shown to users without this status indication.

Employees can access a work dashboard, which displays table assignments, manager information, and a list of employees who report to the user. This information is only shown if relevant to the current employee. An option to update the user's name and banking information is also present. Employees additionally have access to the new game page, which creates play history for a customer. This page presents a form that takes a customer's Driver's License number (scanned in practice), table number, game played, wager amount, and winnings amount. Only tables the current employee works at are displayed, and only the games hosted at the current table are displayed. These pages are exclusive to employees and are not shown to users without this indication.

Managers have access to a work assignment page, where employees are assigned to tables in the casino. New tables and games are created on this page and cannot be deleted, as we are assuming the casino will only expand in the future. Managers can also select which games are played at tables, and this data can be deleted on the games page (discussed later in this section). Finally, this page allows managers to assign the employees they oversee to a table, and these assignments can be deleted on this page. This page is exclusive to managers and is not shown to users without this status indication.

Administrators can access a list of all active people in the casino. They can filter this information by the name of a person and their type, either customer or employee. Each entry is selectable to edit or delete the person's information. This page also allows administrators to create a new user, which can be a customer, employee, or both. Additionally, administrators have access to the statistics page to view casino analytics. The revenue within the past week is shown as a total and in graph form. The revenue from a game is displayed, and each game is selectable from the edit menu. Finally, the most popular games are shown in a pie chart. The 5 most popular games are displayed separately while the remaining games are grouped into an "other" category. These pages are exclusive to administrators and are not sown to users without this indication.

The game list page is available to all users regardless of permissions. This page shows a list of games that can be filtered by name and location. Managers are given an additional option to delete the table-game connection. All users can utilize the dropdown menu at the top-right corner of the webpage to view account information and sign out of Pit Boss. Additionally, if a user does not have access to a certain page, the link will not display in the sidebar menu. If the user attempts to visit a restricted page, they are redirected to a preset 403 error page with options to sign out or return to a valid page.

All data requests are sent to the Java API server using a GET or POST request, depending on whether the data request requires input information. Data is transferred between the API and frontend using JSON key-value pairs and arrays. API requests enter through the ApiController class and call a method in the DatabaseConnector class, which officially makes a request to the Microsoft SQL server. Permissions are handled in the PermissionManager class, which utilizes a properties file with URL paths and permission levels. Driver's License encryption and decryption are handled in the EncryptionManager class.

Backend Discussion

The web app discussed runs through the use of the Pit Boss database and the data contained within it. Tables are established on information related to People, Customers, Employees, Tables, and Games, and these all report their relevant information back to the web app when requested. To further boost security on the back end, only stored procedures and functions are used in order to interact with the tables, so that way input can be properly checked to make sure it is valid. These stored procedures and functions alongside the application account restrictions ensure that no data beyond what is strictly necessary to fulfill a request is sent to the front end, so that way users are not shown what could be private information. As of now, the database currently exists on the Rose-Hulman Titan server within the CSSE department, but it will be removed when the Spring Quarter of the 2022-2023 academic year starts. More information on the various stored procedures and functions, as well as the associated tables, are included in the Database Design section.

Key Challenges

- Challenge: Front End was heavily dependent on functionality already being implemented within the Back End
  Solution: A list of necessary stored procedures was created, and each individual stored procedure was tackled by either Grant Wyness or Aidan Kirk, to allow Canon Maranda to just call said stored procedure from the front end when the time came for said feature to be implemented.
  Analysis: This worked out really well, as many of the stored procedures that Canon would need were already set up in the database, so he could simply read the documentation on each one and create a call to it when a feature was to be implemented on the front end. There were a couple of table-valued functions that were written to complement the stored procedures, as they were much more heavily focused on just obtaining data. When the frontend required more data, such as counting the number of games played, it was simple to update the well-documented stored procedures and functions.
- Challenge: Differing levels of comfort with programming languages used in the solution (particularly, front end vs back end)
  Solution: Although all of the Pit Bosses were comfortable with writing stored procedures and functions in SQL (as well as a connection with Java), Canon was more comfortable with utilizing API calls and the like for the front end than Aidan and Grant. As such, Canon took point on the development of the front end while Aidan and Grant handled writing the necessary stored procedures and functions to support the front end.
  Analysis: This worked out fairly well, but as you may know, front ends usually require a little more time and care than the back end at some points. As such, Canon's efforts each week would require about the same amount of time that Aidan and Grant had to put in combined, just to ensure the connection was smooth, and that the front end looked nice at the same time.
- Challenge: Keeping tables up to date with current definitions
  Solution: Write another SQL script to alter the table definition to include the missing things, then write another script to update the relevant information in the table if we deemed it incorrect afterward.
  Analysis: This solution worked out well – this issue only really began to appear once we reached the end of the project, so we would update the original project database to ensure that the definitions that we wanted worked out well, and then moved the changes over to also be included in the script for generating a new version of the database.

Database Design

The Appendix includes our Entity Relation diagram and Relational Schema as references to the design of our database.

Security Measures

The frontend web application accesses the database through a special account that has carefully crafted permissions within the database – those being reading, writing, and execution permissions. Additionally, the frontend application does not have direct access to the database, so rogue users cannot execute their own queries. Instead, all requests are sent to an API server that sanitizes and routes data to a known stored procedure or function.

The main factor contributing to data integrity is that appropriate administration privileges would only be given to a very select group of people. Customers and employees are both given the respective access to change their data (or add themselves), with customers being limited to just name (since date of birth must be verified and is taken directly from the Driver's License provided). Employees may change personal information, including their name and banking information. Managers can assign their employees to tables. Administrators have access to the data of all users in the database, apart from plaintext Driver's License numbers and banking information. To prevent a breach of privacy of Driver's License numbers, nobody else in the database can view DL numbers associated with someone else– there is no exception for the ability to view these. However, Driver's License numbers may still be used in potential queries that require them; these must come from consenting customers or employees, or the person attempting to run the query must be given explicit permission by the head administrator of the database.

Attackers can potentially reverse engineer our API calls to make unauthorized requests to the database. We mitigate these attacks by requiring a token that represents the current user's driver's license. The server restores the encrypted value to plaintext using a key stored only on the private API server, then sent to the database for validation. Users can have at least one of the following permissions: customer, employee, manager (DL number in the manager column), and administrator (does not have a manager). Data views online and internal command execution are restricted using the above check. In the interest of implementing features for the usability of this application, we do not perform database-level encryption on the DL number field. If this were to be scaled to a larger level, this data would likely experience higher encryption standards. This protocol is secure considering the database and API administrators are reasonable people who will not commit identity theft or fraud.

The banking information of employees is also sensitive and has access restrictions. An employee's account type, routing number, and account number cannot be viewed on the frontend. This specific information is only updatable by the employee themselves or an administrator. Again, we assume the database and API administrators to be reasonable and keep this information otherwise protected. In the interest of implementing features for the usability of this application, we do not perform database-level encryption on the banking information. If this were to be scaled to a larger level, this data would likely experience higher encryption standards. Since our database and internal API services are nearly impossible to access without explicit server credentials and without being at a certain location, any breaches of information will be handled by law enforcement.

Integrity Constraints

There are a handful of referential integrity constraints within the database, which are represented via the arrows on the Relational Schema diagram included in the Appendix of this document. The general rule for update and delete queries that are run on the database is to cascade, with a couple of exceptions for delete. In particular, deleting from person cascades due to the fact that “deleting” is not the default operation – instead, their account is deactivated so that the Casino may continue to track the relevant records with respect to revenue generated, but may not track other information about them unless said person is reactivated. If there is a “hard” delete, then the cascade ensures that the relevant information is removed everywhere. For tables that lack delete stored procedures, their default policy is to reject deletes, since the data they contain should not change that often, and thus doesn’t really warrant a delete.

With that being said, here is a list of the domain integrity constraints in the database:

- Person, Customer, and Employee must have unique DL numbers
- Person.Age must be greater than or equal to 21 and less than the result of GETDATE()
- Person must be either a customer or an employee
- Employee.AccountType must be one of ‘Checking’ or ‘Savings’
- Employees must have a manager (who can be NULL for top-level)
- Plays.PID must be unique (auto-incremented)
- Game.Name must be unique
- Table.Number must be unique (auto-incremented)

Stored Procedures

Create

AddCustomer	Inserts customer into the database with desired driver’s license number, balance, name, and date of birth parameters. Calls AddPerson to insert a person into Customer superclass.
AddEmployee	Inserts employee into the database with desired driver’s license number, date of birth, name, banking information, and manager parameters. Calls AddPerson to insert a person into Employee superclass.
AddGame	Inserts game into the database with name and type parameters.
AddHosts	Inserts an entry into the Hosts table, where the table parameter “hosts” the game parameter.
AddPerson	Inserts a person into the database with desired driver’s license number, name, and date of birth parameters. Called within AddCustomer and AddEmployee stored procedures.
AddPlays	Inserts an entry into the Plays table, where a customer “plays” a game at a table on a certain date. This entry also tracks money won and lost from the transaction.
AddTable	Inserts a table into the database with the location parameter.
AddWorksAt	Inserts an entry into WorksAt table to assign an employee to a table.

Read

GetGameRevenue	Returns the total revenue the casino has made from a game
UserProperties	Determines the owned schemas of a user to see what permissions they have.
UserInformation	Determines if a user exists and their role for login purposes.

Update

ReactivatePerson	Reactivates a person that was soft deleted from the database.
DeactivatePerson	Soft deletes a person, so their information is stored in the database, but the person does not show up in a query of active people.
UpdateCustomerBalance	Updates the balance of a customer based on the modification amount parameter.
UpdateEmployee	Updates the information of an employee using optional parameters.
UpdateGameType	Updates the type of a game.
UpdateName	Updates the name of a customer.
UpdateAges	Updates the age of all people in the database based on the current date.
UpdatePlays	Updates the information for an entry in the Plays table.
UpdateTableLocation	Updates the location for a table.

Delete

DeleteGame	Deletes a game from the database.
DeleteHosts	Deletes an entry from the Hosts table.
DeletePerson	Deletes a person from the database.
DeleteWorksAt	Deletes an employee assignment from a table.

Functions

Table-Valued

fn_EmployeeTables	Retrieves the tables that an employee works at using a driver's license parameter.
fn_GamesTableHosts	Retrieves all of the games a table hosts using a table number parameter.
fn_GetCustomerGames	Retrieves all the games a customer has played.
fn_GetGames	Retrieves all the games that the casino hosts.
fn_GetGamesAndTablesByLocation	Retrieves game and table information for all games with the name parameter played at the location parameter.
fn_GetIndividual	Retrieves all information for a person.
fn_GetLocations	Retrieves all of the locations that tables are located at.
fn_GetPeople	Retrieves all people filtered by name and person type.
fn_GetTablesByLocation	Retrieves all tables located at an area parameter.
fn_getUserPermissions	Retrieves the permissions of all users.
fn_RetrieveEmployeeAssignmentsFromManager	Retrieves all employees and their table assignments that are managed by a person with a driver's license parameter.

fn_RevenueByDates	Retrieves the total revenue of the casino for a date.
fn_GetEmployeesUnderManager	Retrieves all employees managed by a person with a driver's license parameter.

Scalar-Valued

fn_GetTotalRevenue	Returns the total revenue of the casino.

Indexes

There are no user-defined indexes in the Pit Boss database. The store procedures and user-defined functions rely heavily on the primary keys of the tables, so the clustered indexes suffice to efficiently search the database. If the scope of the project is expanded further, the Pit Bosses foresee a useful non-clustered index on driver's licenses in the plays table. Queries that order by driver's license will be sped up if there is a non-clustered index. We attempted to insert user-defined indexes on Name in the Person table and DL number in the Plays table. These are the most commonly used columns for searching outside of primary keys, however, we did not notice a difference in the execution plans. This is because we never return these values by themselves, but instead alongside additional data.

Design Analysis

Strengths

- The program only uses stored procedures and functions to communicate with the database, which makes SQL injection attacks much less likely to occur.
- Stored procedure names, function names, and parameters explain themselves using descriptive nomenclature.
- Integrity constraints are forced upon tables to ensure invalid data are not inserted, removed, or updated.
- Stored procedures and functions relay descriptive error messages back to the front end to ensure easy debugging.
- Functions and procedures are documented and have comments so that any person viewing the database can understand the purpose of a function or procedure.
- Data access and modification on the frontend are only done through a secure API endpoint, strengthening the security of our solution.
- Sensitive information, such as Driver's License numbers and banking details, are not displayed to the end user in plaintext.

Weaknesses

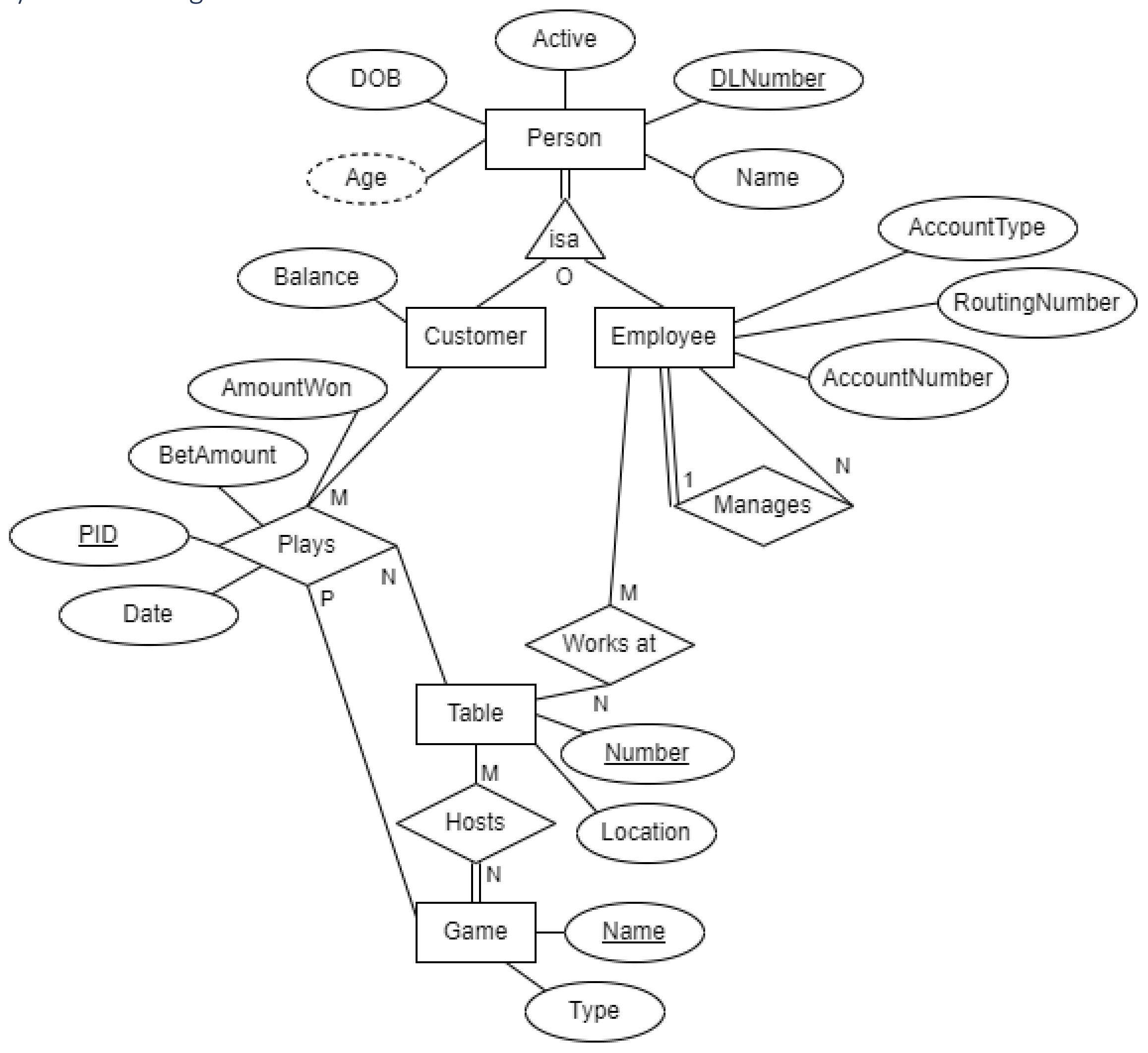
- A deactivated person is never deleted from the database unless manually from the backend. Therefore, if a person never enters the casino again, most of their information is unneeded, but still takes up space in the database. This could be solved by having a "time since deleted" column in the Person table that hard deleted a person from the database after a certain amount of time.
- The ages are not updated for people automatically. The admin has to manually press a button that executes a procedure to update the ages for all people. This functionality would be

unacceptable for a real-world application. This feature would be implemented by running a scheduled task, such as a CRON job, in production.

- Sensitive data is not encrypted within the database. A database administrator can access every user’s DL number and banking information in plaintext, without a decryption key or brute force mechanisms. Instead of managing database-level encryption, we decided to emphasize and expand upon our feature set. We assume the administrators will not expose this information in our scenario, however, this would not be acceptable in a real-world implementation.

Appendix

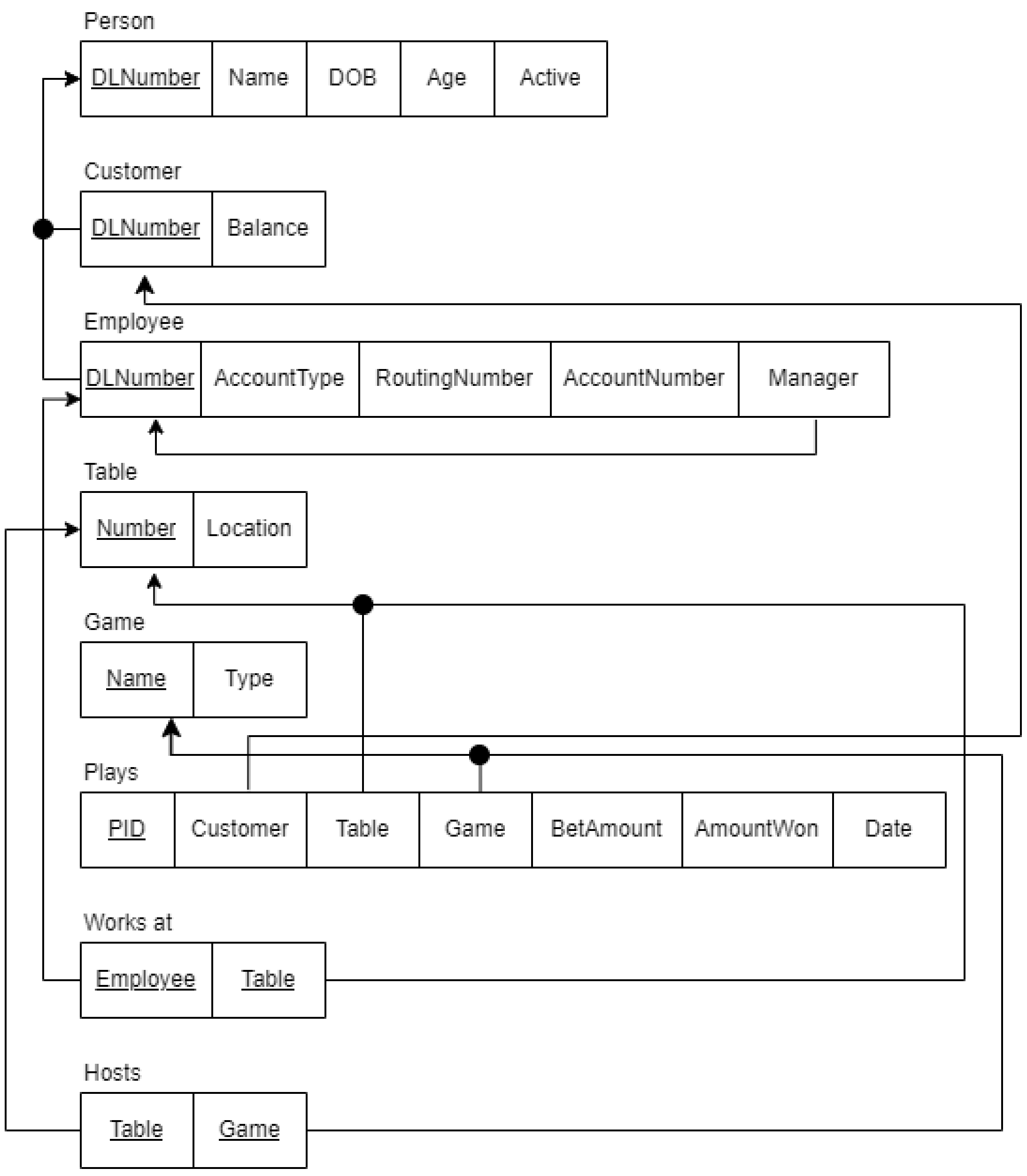
Entity Relation Diagram



Entity Relation Diagram Explanation

Pit Boss is designed to be operated by three types of people: the admin, employees, and managers. A customer plays games at tables via the Plays relation. An employee works at one or many tables via the Works At relation and has a manager via the Manages relation. Employees also enter information about a customer playing a game when a customer plays at their table. A table can host one or many games through the Hosts relation. A manager assigns employees to tables and has access to all other abilities of an employee previously mentioned. The admin is the manager who has no manager above them, and thus can do all things a manager can do. The admin can also create new managers.

Relational Schema



Glossary

API – Application programming interface, a method of executing tasks through web requests and structured data.

Function – Similar functionality to a stored procedure but can return dynamic tables.

Index – The physical ordering of data to allow for search improvements.

Pit Boss – The manager of a casino floor.

Query – One or more sets of SQL code that perform actions on a database.

SQL – Structured query language, provides instructions to the database software.

Stored Procedure – A set of SQL instructions saved on the database that performs tasks based on input and/or output parameters.

UI – User interface, the visual design that encompasses user interaction.

Index

References

Bootstrap Studio – Software to design user interface (HTML/CSS).

Cloudflare Pages – Dynamic hosting for Pit Boss frontend UI.

draw.io – Diagram website used to create ER diagram and schema.

GitHub Codespaces – Virtual machine instance to host Java API.

Import Script Repository – Code for database import script. Accessible by request.

Pit Boss documentation – Initial Problem Statement. Delivered 1/6/2023.

Pit Boss documentation – Security and Data Integrity Analysis Report. Delivered 1/20/2023.

Webserver Repository – Code for frontend and API servers. Accessible by request.