Project Report: *Hotel Reservation System*

Furkan Öztürk
Istanbul Technical University
AI and Data Engineering
ozturkahm20@itu.edu.tr
150200312

Furkan Ünüvar
Istanbul Technical University
AI and Data Engineering
unuvarf20@itu.edu.tr
150200334

Pınar Erçin
Istanbul Technical University
AI and Data Engineering
ercin21@itu.edu.tr
150210336

***Abstract*—This document provides detailed project documentation for *Hotel Reservation System*. It includes an overview of the project idea, ER diagram and data model explanations, CRUD operations, API design, complex query examples, challenges encountered during development, authentication details, and an in-depth look at how Swagger is used for API documentation. The documentation aims to summarize the development process in a professional and clear manner. The source code for this project is available at GitHub Repository.**

I. Overview of the Project Idea

The Hotel Reservation System is designed to manage hotel reservations, customer information, employee details, events, payments, and room services. The system aims to streamline the reservation process, improve the customer experience, and provide a comprehensive management tool for hotel staff. Key features include JWT-based authentication, role-based access control, and detailed API documentation (via Swagger). Postman is used to organize and test all endpoints throughout development.

Furthermore, the system employs the Python `faker` library to generate realistic and random sample data for testing. This approach ensured comprehensive test coverage while removing the need for manually created datasets.

II. ER Diagram and Data Model Explanations

A. ER Diagram

The ER diagram (Fig. 1) illustrates the relationships among entities such as customers, employees, events, reservations, payments, rooms, room services, feedback, and customer-event associations. Each entity is linked by primary and foreign keys to maintain data consistency.

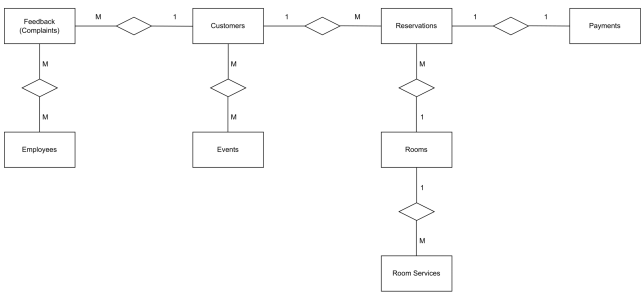

Fig. 1. ER Diagram of the Project.

B. Data Model

The database schema (as defined in the file `hotel_reservation_final.sql`) includes the following tables:

- **customers**:
  - *customer_id* (PK, AUTO_INCREMENT)
  - *name*
  - *phone* (unique)
  - *e_mail* (unique)
- **employees**:
  - *employee_id* (PK, AUTO_INCREMENT)
  - *name*
  - *position*
  - *contact*
- **events**:
  - *event_id* (PK, AUTO_INCREMENT)
  - *event_name*
  - *date*
  - *participation_fee*
- **reservations**:
  - *reservation_id* (PK, AUTO_INCREMENT)
  - *customer_id* (FK references `customers(customer_id)`)
  - *room_id* (FK references `rooms(room_id)`)
  - *check_in_date*
  - *check_out_date*
- **payments**:
  - *payment_id* (PK, AUTO_INCREMENT)
  - *reservation_id* (FK references `reservations(reservation_id)`)
  - *amount*
  - *payment_date*
- **rooms**:
  - *room_id* (PK, AUTO_INCREMENT)
  - *type*
  - *pricing*
  - *capacity*
- **roomservices**:
  - *service_id* (PK, AUTO_INCREMENT)
  - *room_id* (FK references `rooms(room_id)`)
  - *service_type*

  - *cost*
- **feedback**:
  - *feedback_id* (PK, AUTO_INCREMENT)
  - *customer_id* (FK references `customers(customer_id)`)
  - *feedback_details*
  - *feedback_date*
- **customerevents**:
  - *customerevent_id* (PK, AUTO_INCREMENT)
  - *customer_id* (FK references `customers(customer_id)`)
  - *event_id* (FK references `events(event_id)`)
  - *participation_date*

All tables are created in `hotel_reservation_final.sql` with their corresponding FOREIGN KEY constraints for referential integrity.

III. CRUD Operations and Their Implementations

Create, Read, Update, and Delete (CRUD) operations are implemented for each entity in the system. Code snippets below reference the `customers.py` file as an example.

A. Create Operation

```
@app.route('/customers', methods=['POST'])
@jwt_required()
def add_customer():
    data = request.json
    db = get_db_connection()
    ...
    return jsonify({"message": "Customer added
        successfully!"}), 201
```

Listing 1. Example of Create Operation

B. Read Operation

```
@app.route('/customers', methods=['GET'])
@jwt_required()
def get_customers():
    db = get_db_connection()
    ...
    return jsonify(customers)
```

Listing 2. Example of Read Operation

C. Update Operation

```
@app.route('/customers/<int:customer_id>',
    methods=['PUT'])
@jwt_required()
def update_customer(customer_id):
    data = request.json
    db = get_db_connection()
    ...
    return jsonify({"message": "Customer
        updated successfully!"})
```

Listing 3. Example of Update Operation

D. Delete Operation

```
@app.route('/customers/<int:customer_id>',
    methods=['DELETE'])
@jwt_required()
def delete_customer(customer_id):
    db = get_db_connection()
    ...
    return jsonify({"message": "Customer
        deleted successfully!"})
```

Listing 4. Example of Delete Operation

The same pattern applies to entities in `employees.py`, `rooms.py`, `reservations.py`, and other route files.

IV. API Design and Endpoints

The API follows REST principles and employs JWT-based authentication. Endpoints for each entity (e.g., `/customers`, `/rooms`, `/reservations`) are declared in separate files under `routes` or `routes_swagger`. An example base URL is:

http://exampleapi.com/api/v1/

A. Endpoint Examples

- **GET /customers**: Retrieves a list of customers.
- **POST /customers**: Creates a new customer.
- **PUT /customers/:id**: Updates a customer by ID.
- **DELETE /customers/:id**: Deletes a customer by ID.
- Similar endpoints exist for `/employees`, `/rooms`, `/reservations`, `/events`, and `/roomservices`.

V. Postman Collection Usage

The Postman collection is utilized to facilitate efficient testing and collaboration for the Hotel Reservation API. It organizes requests into categories based on entities such as authentication, customers, rooms, reservations, and advanced queries. This organization ensures a structured and logical approach to testing the system's features.

Each request in the collection is pre-configured with example request bodies, token-based authentication, and endpoint details. This setup allows developers and testers to easily validate functionalities, such as secure login, customer management, room operations, reservation handling, and more. The collection also enables the testing of more advanced features, including checking room availability or retrieving complex query results.

Additionally, Postman supports the use of environment variables, simplifying the process of switching between development, staging, and production environments without altering the core request structure. The collection can be shared or exported, ensuring that all team members utilize consistent configurations for testing. By using this collection, the testing process is streamlined, and team collaboration is significantly improved, reducing the chances of errors and ensuring efficient validation of the API's features.

VI. Swagger Documentation

Swagger provides an interactive API documentation interface for the project, simplifying exploration and testing of endpoints. Each resource group (e.g., customers, employees, events) is implemented as a Flask-RESTX Namespace, with clearly defined models and request/response schemas. Endpoints are organized into separate Python files within the `routes_swagger` folder and registered in the main `app_p.py` file.

The Swagger interface supports authentication via Bearer tokens, allowing users to input a token through the "Authorize" button for accessing restricted endpoints. Once the application is running, the Swagger interface can be accessed at:

http://localhost:5000/

A. Key Features

- **Namespace Structure:** Each module (e.g., Customers, Reservations, Payments) has its own namespace, neatly categorized in the Swagger UI. A special *Complex Queries* namespace is added for advanced SQL queries and reports.
- **Authentication:** Endpoints requiring user authentication are secured with JWT (JSON Web Tokens). Instructions for using the "Authorize" button are included in the Swagger UI.
- **Dynamic Endpoints:** For each namespace, RESTful operations like GET, POST, PUT, and DELETE are implemented. For example:
  - Under **Customers**, users can list all customers (admin-only), view their profile, or update information (admin-only).
- **Advanced Queries:** A *Complex Queries* section provides endpoints for SQL queries such as:
  - Listing total reservations per customer.
  - Aggregating employee counts by position.
  - Viewing recent reservations.
- **Interactive Models:** Swagger models describe the request and response payloads, with field requirements and data types automatically validated.
- **User-Friendly Design:** Each endpoint includes detailed input/output information, error codes, and examples, making it easy for developers and testers to interact with the API.

B. Benefits

- Modular design with Flask blueprints ensures scalability.
- JWT-based authentication integrates smoothly with Swagger's security.
- SQL queries provide insights into system data and analytics.

The Swagger integration makes the API user-friendly, enhancing development and testing by offering a clear and interactive interface for understanding and working with the system's features.

```sql
SELECT
    C.name AS Customer_Name,
    COUNT(Res.reservation_id) AS
        Total_Reservations
FROM
    Customers C
LEFT JOIN
    Reservations Res ON C.customer_id = Res.
        customer_id
GROUP BY
    C.name
ORDER BY
    Total_Reservations DESC;
```

Listing 6. Query that lists how many reservations each customer has made

VII. Role-Based Authorization

The system enforces role-based checks for certain operations. Standard users (customers) have limited access, while administrators have full privileges. For instance, in `roomservices.py`, only administrators can add or update a room service:

```
claims = get_jwt()
if claims['role'] != 'admin':
    return {"message": "Access denied"}, 403
```

Listing 5. Admin role check

The same logic applies to employees, rooms, and other administrative resources.

VIII. Complex Query Examples

In this section, examples of complex SQL queries used in the Hotel Reservation System are provided, followed by detailed explanations.

A. Query 1: List of Reservations per Customer

This query retrieves the total number of reservations made by each customer. It performs a LEFT JOIN between the `Customers` table and the `Reservations` table based on the `customer_id` column. The result is grouped by the customer's name and sorted in descending order of the total reservations.

Output Explanation:

- The output includes two columns: `Customer_Name` and `Total_Reservations`.
- For each customer, the query returns the name and the count of reservations made.
- If a customer has no reservations, that customer's name will still appear in the results with a `Total_Reservations` value of `0` due to the LEFT JOIN.

Access Control:

- **Admin Access:** Administrators can view the total reservations for all customers.
- **Customer Access:** Customers can only view their own reservations.

B. Query 2: Number of Employees by Position

```
SELECT
    E.position,
    COUNT(E.employee_id) AS Total_Employees
FROM
    Employees E
GROUP BY
    E.position
ORDER BY
    Total_Employees DESC;
```

Listing 7. Query that lists the number of employees based on their positions

This query calculates the total number of employees grouped by their position within the hotel. It aggregates data from the `Employees` table, groups it by the `position` column, and orders the result in descending order of the employee count.

Output Explanation:

- The output includes two columns: `position` and `Total_Employees`.
- Each row represents a distinct position and the total number of employees holding that position.

Access Control:

- This table is accessible only to administrators.
- Regular customers do not have access to employee-related information.

C. Query 3: Reservations Made in the Last 3 Months

```
SELECT
    C.name AS Customer_Name,
    R.room_type AS Room_Type,
    Res.check_in_date,
    Res.check_out_date
FROM
    Reservations Res
JOIN
    Customers C ON Res.customer_id = C.
        customer_id
JOIN
    Rooms R ON Res.room_id = R.room_id
WHERE
    Res.check_in_date >= DATEADD(MONTH, -3,
        GETDATE())
ORDER BY
    Res.check_in_date DESC;
```

Listing 8. List of all reservations made in the last 3 months

This query retrieves the reservations made within the last three months, including the customer's name, room type, check-in date, and check-out date. It performs JOIN operations between the `Reservations`, `Customers`, and `Rooms` tables to combine the necessary data.

Output Explanation:

- The output includes four columns: `Customer_Name`, `Room_Type`, `check_in_date`, and `check_out_date`.
- Each row represents a reservation made in the last three months, sorted by the check-in date in descending order.

Access Control:

- **Admin Access:** Administrators can view all reservations made in the last three months.
- **Customer Access:** Customers can only view their own reservations that fall within the last three months.

These queries demonstrate how the Hotel Reservation System handles complex data retrieval while maintaining access control to protect sensitive information.

IX. Challenges, Solutions, and Future Work

A. Challenges and Solutions

- **Challenge:** Ensuring database performance under potentially high traffic.
  Solution: Implemented indexes on frequently accessed columns (e.g., *customer_id*, *room_id*) and used efficient queries.
- **Challenge:** Securing endpoints for both administrative and customer roles.
  Solution: Implemented role-based checks (e.g., `claims['role'] != 'admin'`) and used standard JWT libraries.
- **Challenge:** Maintaining consistent API documentation.
  Solution: Used Swagger to document all endpoints and parameters for easy reference.
- **Challenge:** Generating realistic data for testing.
  Solution: Used the Python `faker` library to create sample user, reservation, and event data, ensuring thorough test coverage.

B. Future Work

The application handles user data with two tables:

- **users** (for login credentials): Stores email, hashed passwords, and related account data.
- **customers** (for general hotel records): Stores phone, names, and other profile details.

In many cases, an email used in `users` is also added to `customers` upon sign-up, ensuring a unified record. However, some entries in `customers` may come from phone-based reservations without a matching `users` record.

This design creates a potential risk if an individual reuses another person's email in `customers`. As a solution, an *email verification* process will be implemented to ensure actual email ownership:

- On sign-up, users must confirm a verification link or code.
- This verification step prevents erroneous or malicious email reuse.
- It strengthens consistency between `users` and `customers`.

The current JWT setup can be found in `auth.py` and `app_p.py`. Future changes aim to finalize email verification to address potential vulnerability concerns.

References

[1] Flask Documentation. Available: https://flask.palletsprojects.com/
[2] MySQL Documentation. Available: https://dev.mysql.com/doc/
[3] Project Source Code. Available: https://github.com/Unuvar59/Db_Hotel-Reservation-System