CSI 2132
Databases Deliverable 2

Group No. 80

Gurjot Grewal: 300263760
Ryan Frost-Garant: 300114543
Matthew Petrucci: 300119235

Introduction

This project was done using a PERN (PostgreSQL, Express, React and Node.js) to create the database we have designed in the first deliverable. The objectives were to create a simple functioning SQL database with an easy, beautiful and accessible frontend and implement the features requested for the database on both the clientside and serverside. The project was populated, tested and implemented locally and proven to function well and provide all the prerequisites and features desired for full operation.

This report will be going over the technologies used, how to build and run the database and frontend and the implementations and features used in the database. The project can be found at this GitHub repository link (Golden_Oasis_DB). The project video can be found online here.

Video Timestamps

Requirement	Start Timestamp
1	0:00
2	1:10
3	5:29
4	12:25
5	13:55
6	4:14
7	10:10
8	7:45
9	2:50 14:45

The Application

The utilization of the PERN stack, encompassing PostgreSQL for the database, Express.js for server-side framework, React.js for client-side framework, and Node.js for runtime environment, forms the backbone of modern web application development. Alongside these backend technologies, frontend development benefits from using CSS frameworks such as Tailwind CSS and Bootstrap CSS, which make it easier to create user interfaces that are both responsive and aesthetically pleasing. Additionally, the integration of database management tools like pgAdmin and DBeaver makes it easier to work with PostgreSQL databases efficiently by providing developers with graphical interfaces for activities related to database construction, administration, and querying. With the help of this all-inclusive technology stack, developers can effectively manage database operations, increase development productivity, and create dependable, scalable, and user-friendly online apps.

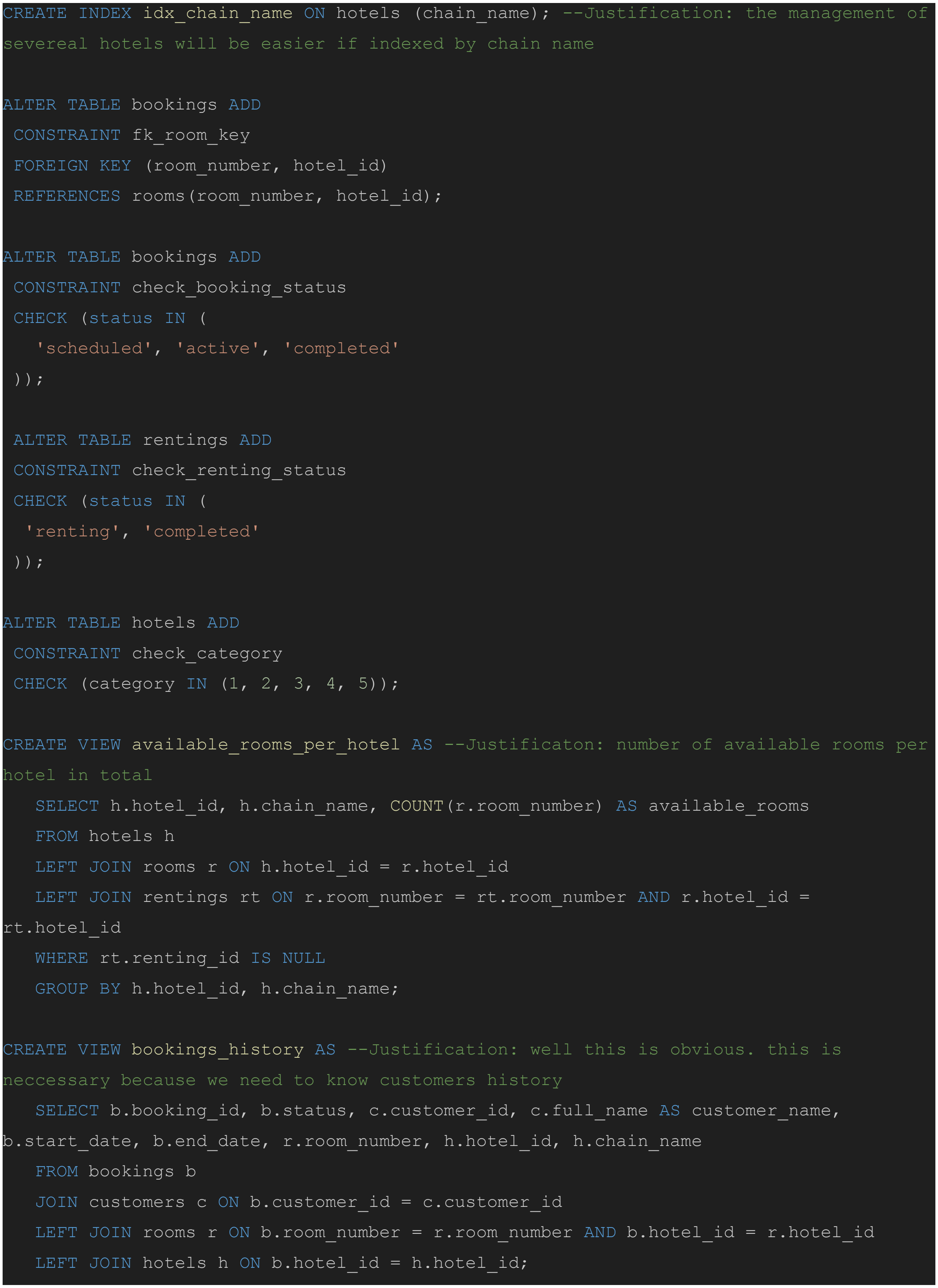
Several programming languages that are necessary for full-stack web application development are included in the PERN stack. The foundation is JavaScript, which is used in the frontend with React.js to create dynamic user interfaces and manage application state, and in the backend with Node.js for server-side scripting and processing HTTP requests. In order to assure data integrity and maximize efficiency, developers can build schemas, edit data, and run queries using SQL, which is essential for dealing with the PostgreSQL database. Web pages' content is organized by HTML and styled and presented by CSS.

Specific steps to guide someone to install your applications:

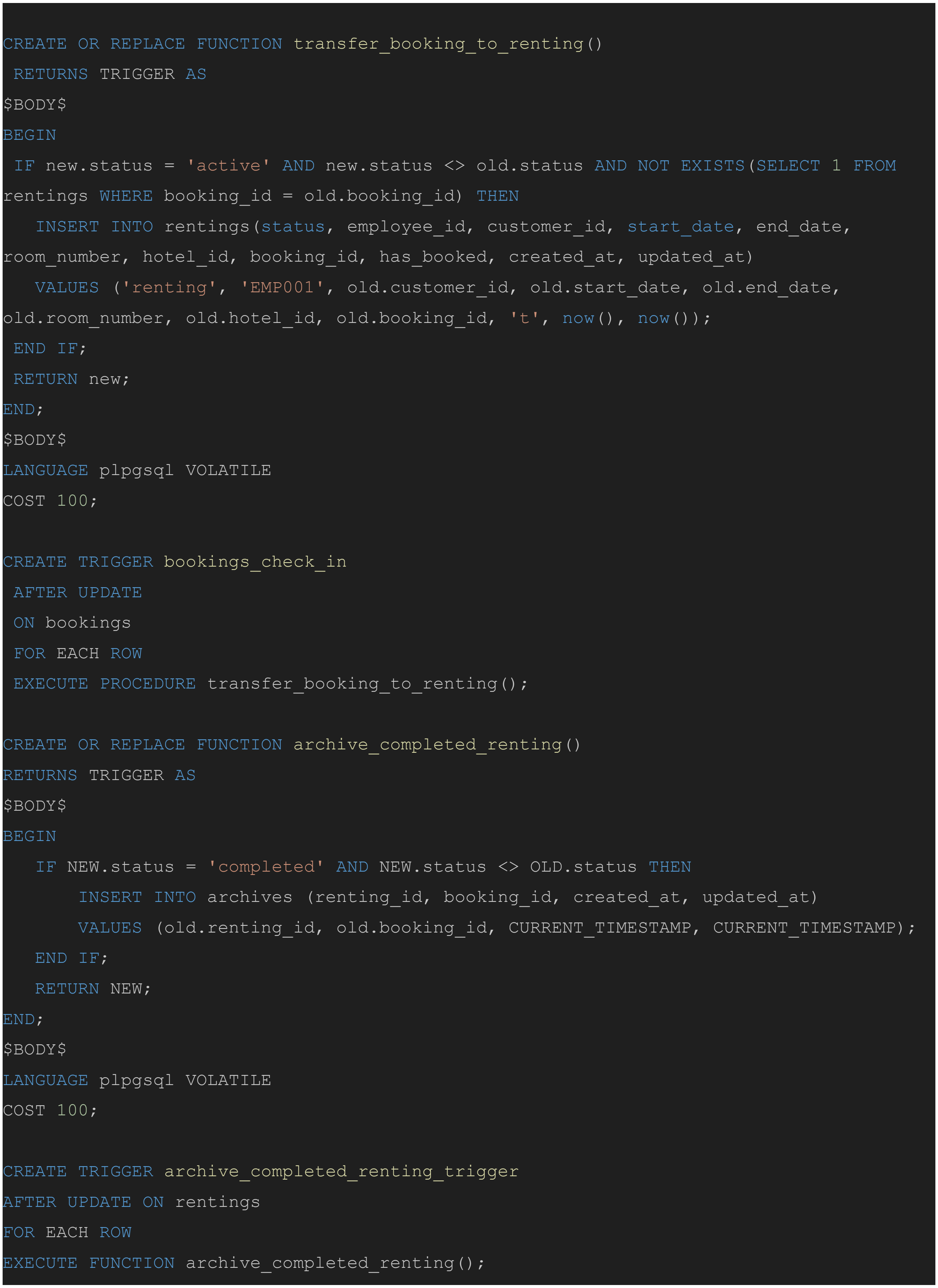
1. Go to the db.js file in the server folder.
    a. Change the password to your postgres password.
2. Go to the server folder in your terminal.
    a. Run the script by using “./script.sh” in your terminal
    b. Continue to enter your postgres password as it created the database.
    c. If you are having trouble running the script then you can enter each command manually in gitbash.
        i. psql -h localhost -U postgres -p 5432 -c "DROP DATABASE IF EXISTS goldenoasisdb;"
        ii. psql -h localhost -U postgres -p 5432 -c "CREATE DATABASE goldenoasisdb;"
        iii. psql -h localhost -d goldenoasisdb -U postgres -p 5432 -f database.sql
    d. Run “node index” in your terminal to start the server. You should see that it has started on port 3001.
3. Go to the client folder in a different terminal while the server is running.
    a. Run npm install.
    b. Run npm start. The website should launch.

```
# Define variables
HOST="localhost"
DATABASE="goldenoasisdb"
USER="postgres"
PORT="5432"
SQL_FILE="database.sql"

# Drop the database if it exists
psql -h $HOST -U $USER -p $PORT -c "DROP DATABASE IF EXISTS $DATABASE;"
```

```sql
# Create the database
psql -h $HOST -U $USER -p $PORT -c "CREATE DATABASE $DATABASE;"

# Run psql command to execute SQL script
psql -h $HOST -d $DATABASE -U $USER -p $PORT < $SQL_FILE

-- Creating Tables
CREATE TABLE addresses(
   street_number INT NOT NULL,
   street_name VARCHAR(100) NOT NULL,
   postal_code VARCHAR(20) NOT NULL,
   city VARCHAR(100) NOT NULL,
   province_state VARCHAR(50),
   created_at TIMESTAMP NOT NULL,
   updated_at TIMESTAMP NOT NULL,
   CONSTRAINT pk_address PRIMARY KEY (street_number, street_name, postal_code)
);

CREATE TABLE hotel_chains(
   chain_name VARCHAR(255) PRIMARY KEY,
   contact_phone_numbers VARCHAR(20) NOT NULL CHECK (contact_phone_numbers ~
'^\+?[0-9\s-]+$'),
   contact_emails VARCHAR(255) NOT NULL,
   created_at TIMESTAMP NOT NULL,
   updated_at TIMESTAMP NOT NULL,
   CONSTRAINT valid_contact_emails CHECK (
       contact_emails ~* '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
   )
);

CREATE TABLE employees(
   employee_id VARCHAR(20) PRIMARY KEY,
   full_name VARCHAR(100) NOT NULL,
   employee_email VARCHAR(255) DEFAULT '' NOT NULL,
   role VARCHAR(50) NOT NULL,
   salary INT NOT NULL,
   street_name VARCHAR(100) NOT NULL,
   street_number INT NOT NULL,
   city VARCHAR(100) NOT NULL,
   province_state VARCHAR(50),
   postal_code VARCHAR(20) NOT NULL,
   created_at TIMESTAMP NOT NULL,
   updated_at TIMESTAMP NOT NULL,
   CONSTRAINT valid_employee_email CHECK (
       employee_email ~* '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
   ),
   FOREIGN KEY(street_number, street_name, postal_code) REFERENCES 
addresses(street_number, street_name, postal_code)
);

CREATE TABLE hotels(
   hotel_id INT NOT NULL,
   chain_name VARCHAR(255) NOT NULL,
   category INT NOT NULL,
   manager_id VARCHAR(20) NOT NULL,
   phone_number VARCHAR(20) NOT NULL CHECK (phone_number ~ '^\+?[0-9\s-]+$'),
   contact_email VARCHAR(255) NOT NULL,
   street_name VARCHAR(100) NOT NULL,
   street_number INT NOT NULL,
   postal_code VARCHAR(20) NOT NULL,
   city VARCHAR(100) NOT NULL,
   province_state VARCHAR(50),
   created_at TIMESTAMP NOT NULL,
   updated_at TIMESTAMP NOT NULL,
   CONSTRAINT valid_contact_email CHECK (
       contact_email ~* '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
   ),
   FOREIGN KEY(chain_name) REFERENCES hotel_chains(chain_name),
   FOREIGN KEY(manager_id) REFERENCES employees(employee_id),
   FOREIGN KEY(street_number, street_name, postal_code) REFERENCES 
addresses(street_number, street_name, postal_code),
   CONSTRAINT pk_hotels PRIMARY KEY (hotel_id, chain_name)
);

CREATE TABLE employee_works_for(
   employee_id VARCHAR(20) NOT NULL,
   hotel_id INT NOT NULL,
   chain_name VARCHAR(255) NOT NULL,

   FOREIGN KEY(employee_id) REFERENCES employees(employee_id),
   FOREIGN KEY(hotel_id, chain_name) REFERENCES hotels(hotel_id, chain_name)
);

CREATE TABLE rooms(
   room_number INT NOT NULL,
   hotel_id INT NOT NULL,
   chain_name VARCHAR(255) NOT NULL,
   price FLOAT NOT NULL,
   capacity INT NOT NULL,
   mountain_view BOOLEAN DEFAULT 'f' NOT NULL,
   sea_view BOOLEAN DEFAULT 'f' NOT NULL,
   is_expandable BOOLEAN DEFAULT 'f' NOT NULL,
   amenities TEXT DEFAULT '',
   damages TEXT DEFAULT '',
   created_at TIMESTAMP NOT NULL,
   updated_at TIMESTAMP NOT NULL,

   FOREIGN KEY(hotel_id, chain_name) REFERENCES hotels(hotel_id, chain_name),
   CONSTRAINT pk_room PRIMARY KEY (room_number, hotel_id)
);

CREATE TABLE customers(
   customer_id SERIAL PRIMARY KEY,
   full_name VARCHAR(100) NOT NULL,
   customer_email VARCHAR(255) DEFAULT '' NOT NULL,
   street_number INT NOT NULL,
   street_name VARCHAR(100) NOT NULL,
   postal_code VARCHAR(20) NOT NULL,
   city VARCHAR(100) NOT NULL,
   province_state VARCHAR(50),
   created_at TIMESTAMP NOT NULL,
   updated_at TIMESTAMP NOT NULL,
   CONSTRAINT valid_customer_email CHECK (
       customer_email ~* '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
   ),
   FOREIGN KEY(street_number, street_name, postal_code) REFERENCES addresses(street_number, street_name, postal_code)
);

CREATE TABLE bookings(
   booking_id SERIAL PRIMARY KEY,
   status VARCHAR(255) NOT NULL,
   customer_id INT,
   start_date TIMESTAMP NOT NULL,
   end_date TIMESTAMP NOT NULL,
   room_number INT NOT NULL,
   hotel_id INT NOT NULL,
   created_at TIMESTAMP NOT NULL,
   updated_at TIMESTAMP NOT NULL,

   FOREIGN KEY(customer_id) REFERENCES customers(customer_id),
   FOREIGN KEY(room_number, hotel_id) REFERENCES rooms(room_number, hotel_id)
);

CREATE TABLE rentings(
   renting_id SERIAL PRIMARY KEY,
   booking_id INT,
   employee_id VARCHAR(20) NOT NULL,
   customer_id INT NOT NULL,
   status VARCHAR(255) NOT NULL,
   start_date TIMESTAMP NOT NULL,
   end_date TIMESTAMP NOT NULL,
   room_number INT NOT NULL,
   hotel_id INT NOT NULL,
   has_booked BOOLEAN DEFAULT 't' NOT NULL,
   created_at TIMESTAMP NOT NULL,
   updated_at TIMESTAMP NOT NULL,

   FOREIGN KEY(customer_id) REFERENCES customers(customer_id),
   FOREIGN KEY(employee_id) REFERENCES employees(employee_id),
   FOREIGN KEY(booking_id) REFERENCES bookings(booking_id),
   FOREIGN KEY(room_number, hotel_id) REFERENCES rooms(room_number, hotel_id)
);

CREATE TABLE archives(
   archive_id SERIAL PRIMARY KEY,
   renting_id INT NOT NULL,
   booking_id INT,
   created_at TIMESTAMP NOT NULL,
   updated_at TIMESTAMP NOT NULL,
   FOREIGN KEY(renting_id) REFERENCES rentings(renting_id),
   FOREIGN KEY (booking_id) REFERENCES bookings(booking_id)
);

CREATE INDEX idx_customer_id ON bookings (customer_id); --Justification: this is very
common in a booking system and will speed up filter queries
CREATE INDEX idx_employee_email ON employees (employee_email); --Justification: for
future systems this will be important for an email system.
```



```
CREATE INDEX idx_chain_name ON hotels (chain_name); --Justification: the management of
severeal hotels will be easier if indexed by chain name

ALTER TABLE bookings ADD
 CONSTRAINT fk_room_key
 FOREIGN KEY (room_number, hotel_id)
 REFERENCES rooms(room_number, hotel_id);

ALTER TABLE bookings ADD
 CONSTRAINT check_booking_status
 CHECK (status IN (
   'scheduled', 'active', 'completed'
 ));

 ALTER TABLE rentings ADD
 CONSTRAINT check_renting_status
 CHECK (status IN (
  'renting', 'completed'
 ));

ALTER TABLE hotels ADD
 CONSTRAINT check_category
 CHECK (category IN (1, 2, 3, 4, 5));

CREATE VIEW available_rooms_per_hotel AS --Justificaton: number of available rooms per
hotel in total
   SELECT h.hotel_id, h.chain_name, COUNT(r.room_number) AS available_rooms
   FROM hotels h
   LEFT JOIN rooms r ON h.hotel_id = r.hotel_id
   LEFT JOIN rentings rt ON r.room_number = rt.room_number AND r.hotel_id =
rt.hotel_id
   WHERE rt.renting_id IS NULL
   GROUP BY h.hotel_id, h.chain_name;

CREATE VIEW bookings_history AS --Justification: well this is obvious. this is
neccessary because we need to know customers history
   SELECT b.booking_id, b.status, c.customer_id, c.full_name AS customer_name,
b.start_date, b.end_date, r.room_number, h.hotel_id, h.chain_name
   FROM bookings b
   JOIN customers c ON b.customer_id = c.customer_id
   LEFT JOIN rooms r ON b.room_number = r.room_number AND b.hotel_id = r.hotel_id
   LEFT JOIN hotels h ON b.hotel_id = h.hotel_id;
```



```
CREATE OR REPLACE FUNCTION transfer_booking_to_renting()
 RETURNS TRIGGER AS
$BODY$
BEGIN
 IF new.status = 'active' AND new.status <> old.status AND NOT EXISTS(SELECT 1 FROM
rentings WHERE booking_id = old.booking_id) THEN
   INSERT INTO rentings(status, employee_id, customer_id, start_date, end_date,
room_number, hotel_id, booking_id, has_booked, created_at, updated_at)
   VALUES ('renting', 'EMP001', old.customer_id, old.start_date, old.end_date,
old.room_number, old.hotel_id, old.booking_id, 't', now(), now());
 END IF;
 RETURN new;
END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;

CREATE TRIGGER bookings_check_in
 AFTER UPDATE
 ON bookings
 FOR EACH ROW
 EXECUTE PROCEDURE transfer_booking_to_renting();

CREATE OR REPLACE FUNCTION archive_completed_renting()
RETURNS TRIGGER AS
$BODY$
BEGIN
   IF NEW.status = 'completed' AND NEW.status <> OLD.status THEN
       INSERT INTO archives (renting_id, booking_id, created_at, updated_at)
       VALUES (old.renting_id, old.booking_id, CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
   END IF;
   RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;

CREATE TRIGGER archive_completed_renting_trigger
AFTER UPDATE ON rentings
FOR EACH ROW
EXECUTE FUNCTION archive_completed_renting();
```