HOTEL MANAGEMENT SYSTEM

GROUP MEMBERS:

1. Pranitha - 112201004
2. Gouthami - 112201003
3. Anusha - 112201043

INTRODUCTION:

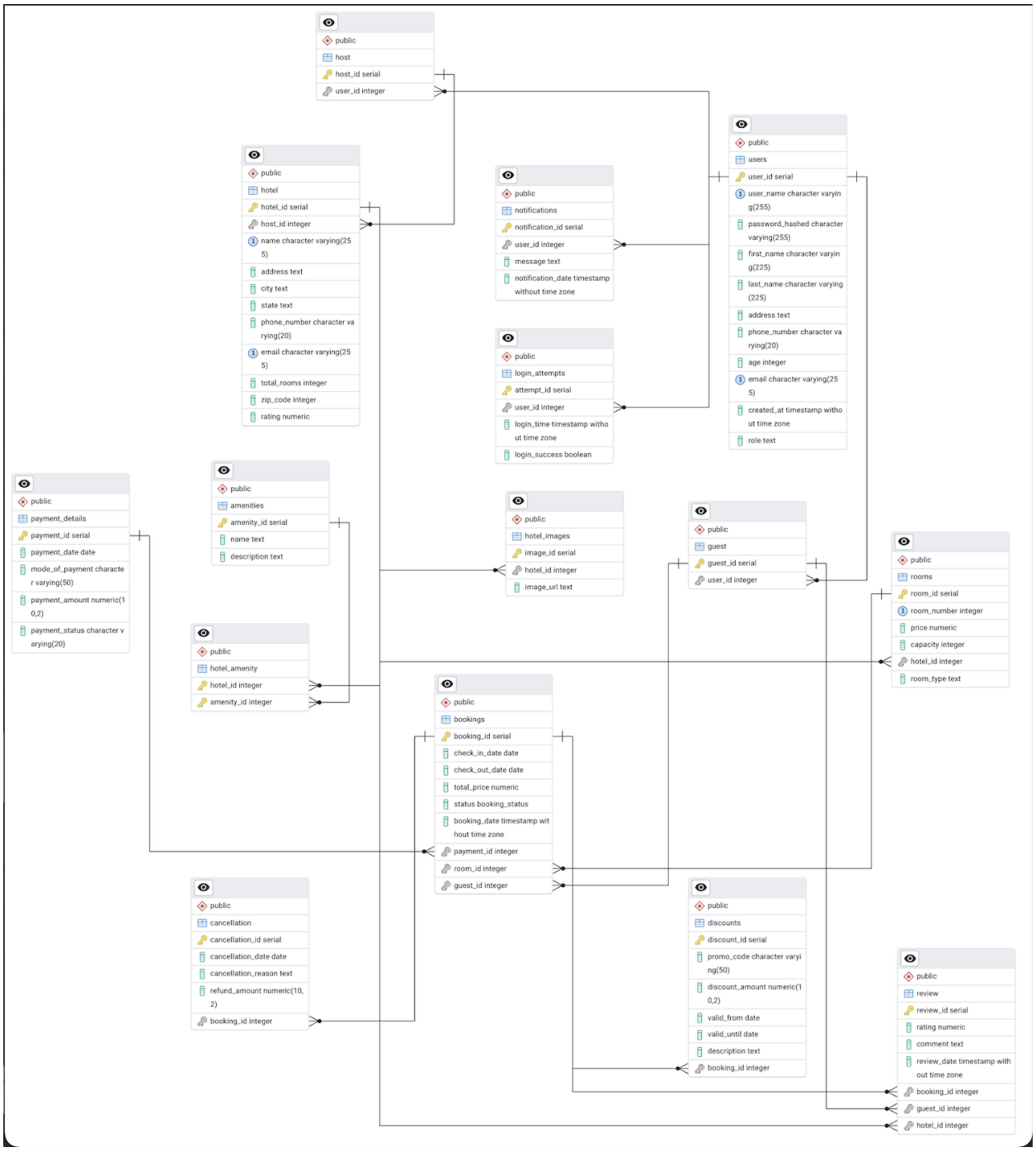
The Hotel Management System is designed to streamline hotel operations, enhance customer experience, and efficiently manage hotel services. It provides an integrated platform for guests and hosts, enabling hotel bookings, property listings, and secure financial transactions.

The system distinguishes between guests and hosts, allowing guests to browse hotels, book stays, and make payments, while hosts can list properties and manage reservations. Secure authentication and role-based access control ensure data protection.

Key features include hotel listing and booking management, tracking check-in/check-out dates, and a review system for guest feedback. A secure payment module ensures smooth transactions, enhancing reliability.

Overall, the system optimizes hotel operations, improves user engagement.

ER DIAGRAM:


public
host
host_id serial
user_id integer
public
hotel
hotel_id serial
host_id integer
name character varying(255)
address text
city text
state text
phone_number character varying(20)
email character varying(255)
total_rooms integer
zip_code integer
rating numeric
public
notifications
notification_id serial
user_id integer
message text
notification_date timestamp without time zone
public
users
user_id serial
user_name character varying(255)
password_hashed character varying(255)
first_name character varying(225)
last_name character varying(225)
address text
phone_number character varying(20)
age integer
email character varying(255)
created_at timestamp without time zone
role text
public
login_attempts
attempt_id serial
user_id integer
login_time timestamp without time zone
login_success boolean
public
payment_details
payment_id serial
payment_date date
mode_of_payment character varying(50)
payment_amount numeric(10,2)
payment_status character varying(20)
public
amenities
amenity_id serial
name text
description text
public
hotel_images
image_id serial
hotel_id integer
image_url text
public
guest
guest_id serial
user_id integer
public
rooms
room_id serial
room_number integer
price numeric
capacity integer
hotel_id integer
room_type text
public
hotel_amenity
hotel_id integer
amenity_id integer
public
bookings
booking_id serial
check_in_date date
check_out_date date
total_price numeric
status booking_status
booking_date timestamp without time zone
payment_id integer
room_id integer
guest_id integer
public
cancellation
cancellation_id serial
cancellation_date date
cancellation_reason text
refund_amount numeric(10,2)
booking_id integer
public
discounts
discount_id serial
promo_code character varying(50)
discount_amount numeric(10,2)
valid_from date
valid_until date
description text
booking_id integer
public
review
review_id serial
rating numeric
comment text
review_date timestamp without time zone
booking_id integer
guest_id integer
hotel_id integer

TABLES:

1. USERS :

Essential for managing user authentication and authorization, differentiating between guests and hosts while securely storing user details.The table ensures secure login with **hashed passwords** and tracks **user registration timestamps**

```
39
40 CREATE TABLE users(
41     user_id SERIAL PRIMARY KEY NOT NULL,
42     user_name VARCHAR(255) UNIQUE NOT NULL,
43     password_hashed VARCHAR(255) NOT NULL,
44     first_name VARCHAR(225) NOT NULL,
45     last_name VARCHAR(225) NOT NULL,
46     address TEXT,
47     phone_number VARCHAR(20),
48     age INTEGER,
49     email VARCHAR(255) UNIQUE,
50     created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
51     role text
52 );
53
54 SELECT * FROM users;
55 -- Enable pgcrypto extension (only required once)
```

Data Output | Messages | Notifications

Showing rows: 1 to 11 | Page No

	user_id [PK] integer	user_name character varying (255)	password_hashed character varying (255)	first_name character varying (225)	last_name character varying (225)	address text	phone_number character varying (20)	age integer	email character varying (255)	created_at timestamp without time zone	role text
1	1	user1	password1	John	Doe	123 Main St, Cityville	111-222-3333	28	user1@example.com	2025-03-01 08:00:00	guest
2	2	user2	password2	Alice	Smith	456 Oak Ave, Townsville	222-333-4444	32	user2@example.com	2025-03-01 08:05:00	host
3	3	user3	password3	Mike	Brown	789 Pine Rd, Villageton	333-444-5555	45	user3@example.com	2025-03-01 08:10:00	guest
4	4	user4	password4	Sarah	Lee	101 Maple Blvd, Hamlet	444-555-6666	27	user4@example.com	2025-03-01 08:15:00	host
5	5	user5	password5	David	Wilson	202 Birch St, Metro City	555-666-7777	36	user5@example.com	2025-03-01 08:20:00	guest
6	6	user6	password6	Emma	Johnson	303 Cedar Dr, Uptown	666-777-8888	29	user6@example.com	2025-03-01 08:25:00	guest
7	7	user7	password7	Liam	Williams	404 Spruce Ln, Downtown	777-888-9999	40	user7@example.com	2025-03-01 08:30:00	guest
8	8	user8	password8	Olivia	Jones	505 Elm St, Suburbia	888-999-0000	33	user8@example.com	2025-03-01 08:35:00	host
9	9	user9	password9	Noah	Davis	606 Walnut Ave, Oldtown	999-000-1111	22	user9@example.com	2025-03-01 08:40:00	guest
10	10	user10	password10	Ava	Martin	707 Chestnut Rd, Riverside	000-111-2222	31	user10@example.com	2025-03-01 08:45:00	guest
11	11	user33	password33	Jp	Doe	123 Main St, Cityville	111-222-3333	28	user33@example.com	2025-03-01 08:00:00	host

2. GUEST :stores information about users who book hotels, linking each guest to a corresponding **user_id** from the **users** table. The **ON DELETE CASCADE** ensures that if a user is deleted, their guest record is also removed, maintaining data integrity.

```sql
CREATE TABLE guest(
    guest_id SERIAL PRIMARY KEY NOT NULL,
    user_id INTEGER NOT NULL  REFERENCES users(user_id) ON DELETE CAS
);
SELECT * FROM guest;
```

Output | Messages | Notifications

guest_id [PK] integer	user_id integer
1	1
2	3
3	5
4	6
5	7
6	9
7	10

3. HOST : stores information about users who list hotels, linking each host to their corresponding **user_id** from the **users** table. The **ON DELETE CASCADE** ensures that if a user is deleted, their host record is also removed, maintaining referential integrity.

```sql
-- Hosts table stores users who list hotels

CREATE TABLE host(
    host_id SERIAL PRIMARY KEY NOT NULL,
    user_id INTEGER NOT NULL  REFERENCES users(user_id) ON DELETE CASCADE
);
SELECT * FROM host;
```

Data Output | Messages | Notifications

	host_id [PK] integer	user_id integer
1	1	2
2	2	4
3	3	8

4. HOTEL :stores details of hotels listed by hosts, including their name, location, contact details, and overall rating. The **host_id** references the **host** table, ensuring that when a host is deleted, their associated hotels are also removed (**ON DELETE CASCADE**), maintaining data consistency.

```sql
CREATE TABLE hotel(
    hotel_id SERIAL PRIMARY KEY NOT NULL,
    host_id INTEGER NOT NULL REFERENCES host(host_id) ON DELETE CASCADE,
    name VARCHAR(255) UNIQUE NOT NULL,
    address TEXT,
    city TEXT,
    state TEXT,
    phone_number VARCHAR(20),
    email VARCHAR(255) UNIQUE,
    total_rooms INTEGER,
    zip_code INTEGER,
    rating DECIMAL
);

SELECT * FROM hotel;
```

Output Messages Notifications

SQL

hotel_id [PK] integer	host_id integer	name character varying (255)	address text	city text	state text	phone_number character varying (20)	email character varying (255)	total_rooms integer	zip_code integer	rating numeric
1	1	Grand Plaza	100 Luxury Ave	Metropolis	State1	111-222-3333	grandplaza@example.com	120	12345	4.
2	2	Sunrise Inn	200 Ocean Dr	Beachside	State2	222-333-4444	sunriseinn@example.com	80	23456	4.
3	3	Mountain Retreat	300 Alpine Rd	Highland	State3	333-444-5555	mountainretreat@example.com	50	34567	4.

5 .ROOMS :stores details of individual hotel rooms, including their number, price, capacity, and availability status. The **hotel_id** reference ensures that each room belongs to a specific hotel, and the **UNIQUE(hotel_id, room_number)** constraint prevents duplicate room numbers within the same hotel.

```sql
CREATE TYPE room_status AS ENUM ('unavailable', 'available');
CREATE TYPE room_type as enum ('AC','Non-AC');

CREATE TABLE Rooms(
    room_id SERIAL PRIMARY KEY NOT NULL,
    room_number INTEGER NOT NULL,
    price DECIMAL,
    capacity INTEGER DEFAULT 1,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    room_type TEXT NOT NULL DEFAULT 'Non-AC',
    -- status room_status DEFAULT 'available',
    hotel_id INTEGER REFERENCES hotel(hotel_id),
    UNIQUE(hotel_id,room_number)

);
ALTER TABLE rooms ADD COLUMN images TEXT[];  -- For PostgreSQL array storage
```

Data Output | Messages | Notifications

	room_id [PK] integer	room_number integer	price numeric	capacity integer	updated_at timestamp without time zone	room_type text	hotel_id integer	images text[]
1	1	101	150.00	2	2025-04-24 15:23:47.554655	AC	1	[null]
2	2	102	170.00	2	2025-04-24 15:23:47.554655	AC	1	[null]
3	3	103	200.00	3	2025-04-24 15:23:47.554655	Non-AC	1	[null]
4	4	201	120.00	2	2025-04-24 15:23:47.554655	AC	2	[null]
5	5	101	130.00	2	2025-04-24 15:23:47.554655	AC	2	[null]
6	6	203	140.00	2	2025-04-24 15:23:47.554655	Non-AC	2	[null]
7	7	301	180.00	3	2025-04-24 15:23:47.554655	AC	3	[null]
8	8	302	190.00	3	2025-04-24 15:23:47.554655	AC	3	[null]
9	9	303	210.00	4	2025-04-24 15:23:47.554655	Non-AC	3	[null]
10	10	304	220.00	4	2025-04-24 15:23:47.554655	Non-AC	3	[null]

6.AMENITIES:stores details about various facilities or services offered in hotel rooms, such as WiFi, air conditioning, or breakfast. Each amenity is uniquely identified by **amenity_id** and includes a **name** and **description** to provide additional details.

```sql
-- Amenities table stores various room amenities

CREATE TABLE Amenities(
    amenity_id SERIAL PRIMARY KEY NOT NULL,
    name TEXT,
    description TEXT
);

SELECT * FROM amenities;
```

Data Output | Messages | Notifications

	amenity_id [PK] integer	name text	description text
1	1	WiFi	High-speed wireless internet
2	2	Breakfast	Complimentary breakfast
3	3	Pool	Outdoor swimming pool
4	4	Gym	Well-equipped fitness center
5	5	Spa	Relaxing spa services
6	6	Parking	Free parking available
7	7	Bar	In-house bar and lounge
8	8	Restaurant	Fine dining restaurant
9	9	Room Service	24/7 in-room dining
10	10	Airport Shuttle	Free shuttle service

Total rows: 10 | Query complete 00:00:00.065

7.HOTEL_AMENITY:establishes a many-to-many relationship between **Hotels** and **Amenities**, linking specific amenities to individual hotel. The **composite primary key**

(hotel_id, amenity_id) ensures that each hotel-amenity pair is unique, preventing duplicate entries.

```
180  ----------------------------------------
181  CREATE TABLE hotel_amenity(
182      hotel_id INTEGER NOT NULL REFERENCES Hotel(hotel_id),
183      amenity_id INTEGER NOT NULL REFERENCES Amenities(amenity_id),
184      PRIMARY KEY (hotel_id, amenity_id)
185  );
186
187
```

Data Output | Messages | Notifications

SQL

	hotel_id [PK] integer	amenity_id [PK] integer
1	1	1
2	1	2
3	1	3
4	2	1
5	2	4
6	2	6
7	3	5
8	3	7
9	3	8
10	4	9
11	4	10

8. BOOKING_STATUS:stores hotel reservation details, including check-in/check-out dates, total price, and booking status. The **guest_id** reference ensures that when a guest is deleted, their associated bookings are also removed (**ON DELETE CASCADE**), maintaining data integrity.

```sql
CREATE TABLE bookings(
    booking_id SERIAL PRIMARY KEY NOT NULL,
    check_in_date DATE NOT NULL,
    check_out_date DATE NOT NULL,
    total_price DECIMAL,
    status booking_status DEFAULT 'pending',
    booking_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    room_id INTEGER REFERENCES Rooms(room_id),
    guest_id INTEGER REFERENCES guest(guest_id) ON DELETE CASCADE
);
ALTER TABLE bookings
  DROP CONSTRAINT bookings_room_id_fkey;

ALTER TABLE bookings
  ADD CONSTRAINT bookings_room_id_fkey
    FOREIGN KEY(room_id)
    REFERENCES rooms(room_id)
    ON DELETE SET NULL;
```

Data Output | Messages | Notifications

	booking_id [PK] integer	check_in_date date	check_out_date date	total_price numeric	status booking_status	booking_date timestamp without time zone	room_id integer	guest_id integer
1	1	2025-04-01	2025-04-05	600.00	confirmed	2025-03-01 12:00:00	1	1
2	2	2025-04-06	2025-04-09	450.00	confirmed	2025-03-02 10:30:00	2	2
3	3	2025-04-10	2025-04-15	750.00	pending	2025-03-03 11:15:00	4	3
4	4	2025-04-16	2025-04-20	500.00	confirmed	2025-03-04 09:45:00	5	4
5	5	2025-04-21	2025-04-25	800.00	confirmed	2025-03-05 14:30:00	7	5
6	6	2025-04-26	2025-04-30	650.00	pending	2025-03-06 15:00:00	8	6
7	7	2025-05-01	2025-05-05	550.00	confirmed	2025-03-07 08:30:00	3	7

9.PAYMENT_DETAILS:stores transaction details for hotel bookings, including the amount, payment method, and status. The **ON DELETE CASCADE** ensures that if a booking or guest is deleted, the associated payment records are also removed, maintaining data consistency.

```sql
CREATE TABLE Payment_Details (
    payment_id SERIAL PRIMARY KEY NOT NULL,
    payment_date DATE ,
    mode_of_payment VARCHAR(50),
    payment_amount NUMERIC(10, 2) NOT NULL,
    payment_status VARCHAR(20),
    booking_id INTEGER REFERENCES Bookings(booking_id) ON DELETE SET NULL
);
-- drop table payment_details CASCADE;
```

a Output Messages Notifications

payment_id [PK] integer	payment_date date	mode_of_payment character varying (50)	payment_amount numeric (10,2)	payment_status character varying (20)	booking_id integer
1	2025-03-01	Credit Card	600.00	confirmed	1
2	2025-03-02	Debit Card	450.00	confirmed	2
3	2025-03-03	Debit card	750.00	pending	3
4	2025-03-04	Credit Card	500.00	confirmed	4
5	2025-03-05	Debit Card	800.00	confirmed	5
6	2025-03-06	Credit card	650.00	pending	6
7	2025-03-07	Credit Card	550.00	confirmed	7
8	2025-03-08	Debit Card	720.00	confirmed	8
9	2025-03-09	Credit Card	830.00	pending	9
10	2025-03-10	Debit card	910.00	confirmed	10

10. REVIEW:The **Review** table stores guest feedback on hotel stays, including ratings, comments, and review dates. It links reviews to specific bookings, guests, and hotels, ensuring that each review is associated with a valid stay experience.

```sql
CREATE TABLE Review(
    review_id SERIAL PRIMARY KEY NOT NULL,
    rating DECIMAL  NOT NULL,
    comment text,
    review_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    booking_id INTEGER REFERENCES Bookings(booking_id),
    guest_id INTEGER REFERENCES guest(guest_id),
    hotel_id INTEGER REFERENCES hotel(hotel_id),
    CONSTRAINT unique_booking_id UNIQUE (booking_id) -- Unique constraint on booking_id

);
```

Output Messages Notifications

SQL

review_id [PK] integer	rating numeric	comment text	review_date timestamp without time zone	booking_id integer	guest_id integer	hotel_id integer
1	4.0	Very comfortable	2025-04-13 10:00:00	2	2	1
2	4.8	Loved the mountain view	2025-03-27 07:30:00	5	5	3
3	4.2	Room was clean and spa…	2025-03-25 10:20:00	7	7	1
4	4.0	Pleasant experience overall	2025-03-29 14:10:00	8	1	2

11.CANCELLATION:records details of canceled bookings, including the cancellation date, reason, and refund amount. The **ON DELETE CASCADE** ensures that if a booking is deleted, its associated cancellation record is also removed, maintaining data integrity.

```sql
-- Cancellation table stores booking cancellation details

CREATE TABLE Cancellation(
    cancellation_id  SERIAL PRIMARY KEY,
    Cancellation_date DATE,
    cancellation_reason TEXT,
    Refund_amount DECIMAL(10, 2),
    booking_id  INTEGER REFERENCES bookings(booking_id) ON DELETE CASCADE
);
```

Data Output | Messages | Notifications

	cancellation_id [PK] integer	cancellation_date date	cancellation_reason text	refund_amount numeric (10,2)	booking_id integer
1	1	2025-03-28	Plans changed	340.00	1
2	2	2025-03-29	Emergency cancellation	450.00	4
3	3	2025-03-30	Customer request	380.00	10

12.DISCOUNTS:stores promotional offers, including promo codes, discount amounts, and validity periods. It links discounts to specific bookings, ensuring that eligible reservations can apply the appropriate discount.

```sql
drop table discounts;
CREATE TABLE Discounts(
    discount_id SERIAL PRIMARY KEY NOT NULL,
    Promo_code VARCHAR(50) NOT NULL,
    discount_amount NUMERIC(10, 2) NOT NULL,
    valid_from DATE NOT NULL,
    valid_until DATE NOT NULL,
    description TEXT,
    booking_id  INTEGER REFERENCES bookings(booking_id) ON DELETE CASCADE
);
SELECT * FROM discounts;
```

Data Output | Messages | Notifications

	discount_id [PK] integer	promo_code character varying (50)	discount_amount numeric (10,2)	valid_from date	valid_until date	description text	booking_id integer
1	1	DISCOUNT10	10.00	2025-04-01	2025-04-30	Spring discount	[null]
2	2	SUMMER15	15.00	2025-06-01	2025-06-30	Summer special	[null]
3	3	WINTER20	20.00	2025-12-01	2025-12-31	Winter sale	[null]
4	4	FLASH5	5.00	2025-03-15	2025-03-20	Flash deal	[null]
5	5	NEWYEAR25	25.00	2026-01-01	2026-01-31	New Year offer	[null]
6	6	VIP30	30.00	2025-05-01	2025-05-31	VIP discount	[null]
7	7	EARLYBIRD10	10.00	2025-07-01	2025-07-15	Early bird offer	[null]
8	8	LASTMIN20	20.00	2025-08-01	2025-08-10	Last minute deal	[null]
9	9	EXCLUSIVE15	15.00	2025-09-01	2025-09-30	Exclusive offer	[null]
10	10	SEASONAL5	5.00	2025-03-01	2025-03-31	Seasonal discount	[null]

13.NOTIFICATIONS:stores messages sent to users, such as booking confirmations, payment updates, or promotional alerts. The **ON DELETE CASCADE** ensures that if a user is deleted, their associated notifications are also removed, maintaining data consistency.

```sql
CREATE TABLE notifications (
    notification_id SERIAL PRIMARY KEY NOT NULL,
    user_id INTEGER REFERENCES users(user_id) ON DELETE CASCADE,
    message TEXT NOT NULL,
    notification_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
SELECT * FROM notifications;
```

Output | Messages | Notifications

notification_id [PK] integer	user_id integer	message text	notification_date timestamp without time zone
1	1	Bookings are open now—reserve your stay today!	2025-03-01 12:05:00
2	2	New discounts available on your next booking.	2025-03-01 12:10:00
3	3	Enjoy exclusive deals—book your room now!	2025-03-02 09:15:00
4	4	Hurry! Special rates available for a limited time.	2025-03-02 10:20:00
5	5	Reserve your room and save with our latest offers.	2025-03-03 11:05:00
6	6	Secure your reservation; bookings are officially open!	2025-03-03 12:30:00
7	7	Check out our new offers and discount promotions.	2025-03-04 08:45:00
8	8	Experience luxury at affordable prices—book today!	2025-03-04 09:50:00
9	9	Limited time discount on premium rooms—act fast!	2025-03-05 14:30:00
10	10	Don't miss out—special booking rates available now.	2025-03-05 15:20:00
11	1	Exclusive offer: Book now and enjoy extra perks.	2025-04-01 12:05:00
12	2	Discover new deals: Discounts available on selected hotels.	2025-04-01 12:10:00

14.LOGIN_ATTEMPTS:tracks user login activity, recording the time of each attempt and whether it was successful. This helps in monitoring security, detecting unauthorized access attempts, and implementing account lockout policies if needed.

```sql
CREATE TABLE Login_Attempts (
    attempt_id SERIAL PRIMARY KEY,
    user_id INT REFERENCES users(user_id),
    login_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    login_success BOOLEAN
);
```

Data Output | Messages | Notifications

	attempt_id [PK] integer	user_id integer	login_time timestamp without time zone	login_success boolean
1	1	1	2025-03-01 07:55:00	true
2	2	2	2025-03-01 08:00:00	true
3	3	3	2025-03-01 08:05:00	false
4	4	4	2025-03-01 08:10:00	true
5	5	5	2025-03-01 08:15:00	true
6	6	6	2025-03-01 08:20:00	false
7	7	7	2025-03-01 08:25:00	true
8	8	8	2025-03-01 08:30:00	true
9	9	9	2025-03-01 08:35:00	false
10	10	10	2025-03-01 08:40:00	true
11	11	3	2025-04-01 08:05:00	true
12	12	4	2025-04-01 08:10:00	false
13	13	5	2025-04-01 08:15:00	false

Total rows: 14 Query complete 00:00:00.082

All integrity constraints and general constraints with a brief description:

Primary Keys:

1. `users`: `user_id`
2. `guest`: `guest_id`
3. `host`: `host_id`
4. `hotel`: `hotel_id`
5. `rooms`: `room_id`
6. `amenities`: `amenity_id`
7. `bookings`: `booking_id`
8. `payment_details`: `payment_id`
9. `review`: `review_id`
10. `cancellation`: `cancellation_id`
11. `discounts`: `discount_id`
12. `notifications`: `notification_id`
13. `login_attempts`: `attempt_id`
14. `hotel_amenity`: Composite primary key (`hotel_id`, `amenity_id`).

Foreign Keys

1. `guest`:

   `user_id` references `users(user_id)` with `ON DELETE CASCADE`.
2. `host`:

   `user_id` references `users(user_id)` with `ON DELETE CASCADE`.
3. `hotel`:

   `host_id` references `host(host_id)` with `ON DELETE CASCADE`.
4. `rooms`:

   `hotel_id` references `hotel(hotel_id)` with `ON DELETE SET NULL`.
5. `bookings`:

   `room_id` references `rooms(room_id)` with `ON DELETE SET NULL`.

   `guest_id` references `guest(guest_id)` with `ON DELETE CASCADE`.
6. `payment_details`:

booking_id references bookings(booking_id) with ON DELETE SET NULL.

7. review:

   booking_id references bookings(booking_id).

   guest_id references guest(guest_id).

   hotel_id references hotel(hotel_id).

8. cancellation:

   booking_id references bookings(booking_id) with ON DELETE CASCADE.

9. notifications:

   user_id references users(user_id) with ON DELETE CASCADE.

10. login_attempts:

    user_id references users(user_id).

11. hotel_amenity:

    hotel_id references hotel(hotel_id).

    amenity_id references amenities(amenity_id).

Unique Constraints

1. users:
   - user_name must be unique.
   - Email must be unique.
2. hotel:
   - the name must be unique.
   - Email must be unique.
3. rooms:
   - room_number must be unique within the same hotel (hotel_id, room_number uniqueness).
4. review:
   - booking_id must be unique (one review per booking).

NOT NULL Constraints

1. `users`:

   `user_id`, `user_name`, `password_hashed`, `first_name`, `last_name`, `created_at`, and `role` cannot be null.
2. `guest`:

   `guest_id` and `user_id` cannot be null.
3. `host`:

   `host_id` and `user_id` cannot be null.
4. `hotel`:

   `hotel_id`, `host_id`, `name`, `address`, `city`, `state`, `phone_number`, `email`, `total_rooms`, `zip_code`, and `rating` cannot be null.
5. `rooms`:

   `room_id`, `room_number`, `price`, `hotel_id`, and `room_type` cannot be null.
6. `bookings`:

   `booking_id`, `check_in_date`, `check_out_date`, `total_price`, `booking_date`, and `guest_id` cannot be null.

ENUM Types

1. `bookings.status`:

   Restricted to `pending`, `confirmed`, or `cancelled`.
2. `rooms.room_type`:

   Restricted to `AC` or `Non-AC`.

Default Values

1. `users.created_at`:

   Defaults to `CURRENT_TIMESTAMP`.
2. `rooms.capacity`:

Defaults to 1.

3. `bookings.booking_date`:

   Defaults to `CURRENT_TIMESTAMP`.

Composite Keys

1. `hotel_amenity`:
   - Primary key combines `hotel_id` and `amenity_id`.

Cascading Deletes

1. Deleting a `user` automatically deletes linked `guest` or `host` entries.
2. Deleting a `guest` automatically deletes their `bookings`.
3. Deleting a `host` automatically deletes their `hotel` entries.
4. Deleting a `hotel` automatically sets `rooms.hotel_id` to `NULL`.

Other Constraints

1. `rooms.room_type`:

   Explicitly defined as `TEXT` but behaves like an enum (`AC/Non-AC`).

2. `users.role`:

   Implicitly expected to be `guest`, `host`, or `admin` (though not enforced via `ENUM`).

3. `review.rating`:

   Defined as `DECIMAL` but typically validated as a value between `0` and `5` (application-level).

VIEWS:

For Admin:

1.v_system_stats: This view that summarizes key system statistics, including total users, bookings, revenue, hotels, and rooms.

```
1060  CREATE OR REPLACE VIEW v_system_stats AS
1061  SELECT
1062    (SELECT COUNT(*) FROM users)                                    AS total_users,
1063    (SELECT COUNT(*) FROM bookings)                                 AS total_bookings,
1064    (SELECT COALESCE(SUM(payment_amount),0)
1065       FROM payment_details
1066      WHERE payment_status = 'completed')                            AS total_revenue,
1067    (SELECT COUNT(*) FROM hotel)                                    AS total_hotels,
1068    (SELECT COUNT(*) FROM rooms)                                    AS total_rooms;
1069
1070  SELECT * FROM v_system_stats;
1071  -- 2) Recent bookings (last 5)
```

Data Output | Messages | Notifications

	total_users bigint	total_bookings bigint	total_revenue numeric	total_hotels bigint	total_rooms bigint
1	14	10	0	4	10

2.v_recent_bookings: Creates a view that shows details of the 5 most recent bookings, including guest name, hotel name, booking dates,status, and price.

```sql
CREATE OR REPLACE VIEW v_recent_bookings AS
SELECT b.booking_id,
       b.check_in_date,
       b.check_out_date,
       b.total_price,
       b.status,
       b.booking_date,
       u.user_name    AS guest_user,
       h.name         AS hotel_name
  FROM bookings b
  JOIN guest g    ON b.guest_id = g.guest_id
  JOIN users u    ON g.user_id   = u.user_id
  JOIN rooms r    ON b.room_id   = r.room_id
  JOIN hotel h    ON r.hotel_id  = h.hotel_id
 ORDER BY b.booking_date DESC
 LIMIT 5;
 SELECT * FROM v_recent_bookings;

 --ADMIN FUNCTIONS AND TRIGGERS

 -- 1) cancel a single booking: update status → insert cancellation → delete review
CREATE OR REPLACE FUNCTION cancel_booking(
```

Data Output | Messages | Notifications

	booking_id integer	check_in_date date	check_out_date date	total_price numeric	status booking_status	booking_date timestamp without time zone	guest_user character varying (255)	hotel_name character varying (255)
1	10	2025-05-16	2025-05-20	910.00	confirmed	2025-03-10 10:00:00	user5	Mountain Retreat
2	9	2025-05-11	2025-05-15	830.00	pending	2025-03-09 09:15:00	user3	Mountain Retreat
3	8	2025-05-06	2025-05-10	720.00	cancelled	2025-03-08 11:00:00	user1	Sunrise Inn
4	7	2025-05-01	2025-05-05	550.00	confirmed	2025-03-07 08:30:00	user10	Grand Plaza
5	6	2025-04-26	2025-04-30	650.00	pending	2025-03-06 15:00:00	user9	Mountain Retreat

For Guests:

1.v_guests_upcoming_travels*:This is a view that lists all upcoming stays for guests (non-cancelled and with future check-in dates), including hotel, room, booking, and payment details.*

```sql

-- 1) Upcoming stays (non-cancelled, check_in >= today)
CREATE OR REPLACE VIEW v_guest_upcoming_travels AS
SELECT
  b.booking_id,
  g.user_id,
  b.check_in_date,
  b.check_out_date,
  b.total_price,
  b.status,
  r.room_number,
  h.name       AS hotel_name,
  pd.payment_status
FROM bookings b
JOIN guest g    ON b.guest_id = g.guest_id
JOIN rooms r    ON b.room_id = r.room_id
JOIN hotel h    ON r.hotel_id = h.hotel_id
LEFT JOIN payment_details pd ON b.booking_id = pd.booking_id
WHERE b.status <> 'cancelled'
  AND b.check_in_date >= CURRENT_DATE
ORDER BY b.check_in_date;


```

Data Output | Messages | Notifications

```
CREATE VIEW

Query returned successfully in 60 msec.
```

2.v_guest_travel_history: Creates a view that displays a guest's past travel history (excluding cancelled bookings) with hotel, room, booking, and payment details.

```sql

-- 2) Past stays (non-cancelled, check_out < today)
CREATE OR REPLACE VIEW v_guest_travel_history AS
SELECT
  b.booking_id,
  g.user_id,
  b.check_in_date,
  b.check_out_date,
  b.total_price,
  b.status,
  r.room_number,
  h.name AS hotel_name,
  pd.payment_status
FROM bookings b
JOIN guest g    ON b.guest_id = g.guest_id
JOIN rooms r    ON b.room_id = r.room_id
JOIN hotel h    ON r.hotel_id = h.hotel_id
LEFT JOIN payment_details pd ON b.booking_id = pd.booking_id
WHERE b.status <> 'cancelled'
  AND b.check_out_date < CURRENT_DATE
ORDER BY b.check_out_date DESC;

```

Data Output | Messages | Notifications

```
CREATE VIEW

Query returned successfully in 54 msec.
```

3.v_guest_cancellation_history: Creates a view that shows guests' cancellation history, including booking details, hotel info, cancellation reason, date, and refund amount.

```
1219  ORDER BY b.check_out_date DESC;
1220
1221  -- 3) Cancellation history
1222  CREATE OR REPLACE VIEW v_guest_cancellation_history AS
1223  SELECT
1224    c.cancellation_id,
1225    g.user_id,
1226    b.booking_id,
1227    c.cancellation_date,
1228    c.cancellation_reason,
1229    c.refund_amount,
1230    r.room_number,
1231    h.name AS hotel_name
1232  FROM cancellation c
1233  JOIN bookings b  ON c.booking_id = b.booking_id
1234  JOIN guest g     ON b.guest_id    = g.guest_id
1235  JOIN rooms r     ON b.room_id     = r.room_id
1236  JOIN hotel h     ON r.hotel_id    = h.hotel_id
1237  ORDER BY c.cancellation_date DESC;
1238
1239
```

Data Output | Messages | Notifications

```
CREATE VIEW

Query returned successfully in 97 msec.
```

For Host:

1.v_rooms_with_booking_counts: Creates a view that lists all rooms along with their booking count, showing room details and total number of times each room has been booked.

```sql
CREATE OR REPLACE VIEW v_rooms_with_booking_counts AS
SELECT
  r.room_id,
  r.hotel_id,
  r.room_number,
  r.price,
  r.capacity,
  r.images,
  r.updated_at,
  r.room_type,
  COALESCE(bc.total_bookings, 0) AS total_bookings
FROM rooms r
LEFT JOIN (
  SELECT room_id, COUNT(*) AS total_bookings
    FROM bookings
   GROUP BY room_id$
) bc ON r.room_id = bc.room_id;
```

Data Output | Messages | Notifications

```
CREATE VIEW

Query returned successfully in 52 msec.
```

Functions:

For Admin:

1.cancel_booking: This function provides an admin to handle booking cancellations, ensuring proper logging, refund processing, and cleanup of related data.

```
1089
1090    --ADMIN FUNCTIONS AND TRIGGERS
1091
1092    -- 1) cancel a single booking: update status → insert cancellation → delete review
1093  CREATE OR REPLACE FUNCTION cancel_booking(
1094      p_booking_id  INTEGER,
1095      p_reason      TEXT
1096  ) RETURNS VOID LANGUAGE plpgsql AS $$
1097  DECLARE
1098      v_price  NUMERIC;
1099  BEGIN
1100      -- grab the amount so we can refund
1101      SELECT total_price INTO v_price
1102        FROM bookings
1103       WHERE booking_id = p_booking_id;
1104
1105      -- mark it cancelled
1106      UPDATE bookings
1107         SET status = 'cancelled'
1108       WHERE booking_id = p_booking_id;
1109
1110      -- log into cancellation
1111      INSERT INTO cancellation(
1112          cancellation_date,
1113          cancellation_reason,
1114          refund_amount,
1115          booking_id
1116      ) VALUES (
1117          CURRENT_DATE,
1118          p_reason,
1119          v_price,
1120          p_booking_id
1121      );
1122
1123      -- drop any review attached
1124      DELETE FROM review
1125       WHERE booking_id = p_booking_id;
1126  END;
1127  $$;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 101 msec.
```

2.cancel_bookings_by_hotel: Creates an admin function to batch-cancel all active bookings for a given hotel by invoking the single-booking cancel helper for each, ensuring proper logging and refunds.

```
1128
1129  -- 2) cancel *all* bookings for a given hotel_id
1130  CREATE OR REPLACE FUNCTION cancel_bookings_by_hotel(
1131      p_hotel_id  INTEGER,
1132      p_reason    TEXT
1133  ) RETURNS VOID LANGUAGE plpgsql AS $$
1134  DECLARE
1135      rec  RECORD;
1136  BEGIN
1137      FOR rec IN
1138        SELECT b.booking_id
1139          FROM bookings b
1140          JOIN rooms r ON b.room_id = r.room_id
1141         WHERE r.hotel_id = p_hotel_id
1142           AND b.status   <> 'cancelled'
1143      LOOP
1144        -- call the single-booking helper
1145        PERFORM cancel_booking(rec.booking_id, p_reason);
1146      END LOOP;
1147  END;
1148  $$;
1149
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 48 msec.
```

For Guest:

1.fn_get_guest_id: Creates a function to retrieve the guest_id associated with a given user_id.

```
1238
1239
1240  -- ———————— HELPER FUNCTIONS ————————
1241
1242  -- A) look up guest_id from user_id
1243  CREATE OR REPLACE FUNCTION fn_get_guest_id(p_user_id INT)
1244  RETURNS INT LANGUAGE sql AS $$
1245    SELECT guest_id FROM guest WHERE user_id = p_user_id;
1246  $$;
1247
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 57 msec.
```

2.fn_guest_upcoming: Creates a function that returns all upcoming stays for a given user_id, pulling booking, room, hotel, and payment details from the v_guest_upcoming_travels view.

```sql
-- B) upcoming stays by user_id
CREATE OR REPLACE FUNCTION fn_guest_upcoming(p_user_id INT)
RETURNS TABLE(
  booking_id       INT,
  user_id          INT,
  check_in_date    DATE,
  check_out_date   DATE,
  total_price      NUMERIC,
  status           TEXT,
  room_number      INT,
  hotel_name       TEXT,
  payment_status   TEXT
) LANGUAGE sql AS $$
  SELECT *
    FROM v_guest_upcoming_travels
   WHERE user_id = p_user_id;
$$;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 58 msec.
```

3.fn_guest_past: Creates a function that returns all past (non-cancelled) stays for a given user_id, including booking, room, hotel, and payment details from the v_guest_travel_history view.

```sql

-- C) past stays by user_id
CREATE OR REPLACE FUNCTION fn_guest_past(p_user_id INT)
RETURNS TABLE(
  booking_id       INT,
  user_id          INT,
  check_in_date    DATE,
  check_out_date   DATE,
  total_price      NUMERIC,
  status           TEXT,
  room_number      INT,
  hotel_name       TEXT,
  payment_status   TEXT
) LANGUAGE sql AS $$
  SELECT *
    FROM v_guest_travel_history
   WHERE user_id = p_user_id;
$$;

```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 58 msec.
```

4.fn_guest_cancellations: Creates a function that returns all cancellation records for a given user_id, including cancellation details, room number, and hotel information from the v_guest_cancellation_history view.

```sql
-- D) cancellations by user_id
CREATE OR REPLACE FUNCTION fn_guest_cancellations(p_user_id INT)
RETURNS TABLE(
  cancellation_id   INT,
  user_id           INT,
  booking_id        INT,
  cancellation_date DATE,
  cancellation_reason TEXT,
  refund_amount     NUMERIC,
  room_number       INT,
  hotel_name        TEXT
) LANGUAGE sql AS $$
  SELECT *
    FROM v_guest_cancellation_history
   WHERE user_id = p_user_id;
$$;

```

Data Output Messages Notifications

```
CREATE FUNCTION

Query returned successfully in 45 msec.
```

5.fn_get_room_details: Creates a function that, given a room ID, fetches the room's details along with its hotel's info, average rating, and list of amenities.

```sql

CREATE OR REPLACE FUNCTION fn_get_room_details(p_room_id INT)
RETURNS TABLE (
  room_id        INT,
  room_number    INT,
  price          NUMERIC,
  capacity       INT,
  images         TEXT[],
  updated_at     TIMESTAMP,
  room_type      TEXT,
  hotel_id       INT,
  hotel_name     TEXT,
  address        TEXT,
  city           TEXT,
  state          TEXT,
  hotel_phone    TEXT,
  hotel_email    TEXT,
  avg_rating     NUMERIC,
  amenities      TEXT[]
)
LANGUAGE sql AS $$
SELECT
  r.room_id,
  r.room_number,
  r.price,
  r.capacity,
  r.images,
  r.updated_at,
  r.room_type,
  h.hotel_id,
  h.name          AS hotel_name,
  h.address,
  h.city,
  h.state,
  h.phone_number  AS hotel_phone,
  h.email         AS hotel_email,
  COALESCE((
    SELECT AVG(rv.rating)
      FROM review rv
     WHERE rv.hotel_id = h.hotel_id
  ), 0.0)           AS avg_rating,
  COALESCE((
    SELECT array_agg(a.name)
      FROM hotel_amenity ha
      JOIN amenities a ON ha.amenity_id = a.amenity_id
     WHERE ha.hotel_id = h.hotel_id
  ), ARRAY[]::TEXT[]) AS amenities
FROM rooms r
JOIN hotel h ON r.hotel_id = h.hotel_id
WHERE r.room_id = p_room_id;
$$;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 50 msec.
```

6.get_available_rooms_by_location: This function helps in finding available hotel rooms in a specific city based on the preferences. we can filter by price range, room capacity, and hotel rating. If you provide check-in and check-out dates, it ensures the room isn't already booked during that period. The results show room details like hotel name, room number, price, capacity, and rating, sorted from cheapest to most expensive. If the dates are not specified, it just lists rooms that meet the other criteria.

```
624
625 CREATE OR REPLACE FUNCTION get_available_rooms_by_location(
626     p_city TEXT,
627     p_check_in_date DATE DEFAULT NULL,
628     p_check_out_date DATE DEFAULT NULL,
629     p_min_price DECIMAL DEFAULT 0,
630     p_max_price DECIMAL DEFAULT 99999,
631     p_min_capacity INTEGER DEFAULT 1,
632     p_min_rating DECIMAL DEFAULT 0
633 )
634 RETURNS TABLE(
635     room_id INTEGER,
636     hotel_name TEXT,
637     room_number INTEGER,
638     room_type TEXT,
639     price DECIMAL,
640     capacity INTEGER,
641     hotel_rating DECIMAL
642 ) AS $$
643 BEGIN
644     RETURN QUERY
645     SELECT
646         r.room_id,
647         h.name::TEXT AS hotel_name, -- Cast to TEXT
648         r.room_number,
649         r.room_type::TEXT AS room_type, -- Cast to TEXT
650         r.price,
651         r.capacity,
652         h.rating
653     FROM rooms r
654     INNER JOIN hotel h ON r.hotel_id = h.hotel_id
655     WHERE
656         lower(trim(h.city)) = lower(trim(p_city))
657     AND COALESCE(r.price, 0) BETWEEN p_min_price AND p_max_price
658     AND COALESCE(r.capacity, 0) >= p_min_capacity
659     AND COALESCE(h.rating, 0) >= p_min_rating
660         AND (
661             p_check_in_date IS NULL OR p_check_out_date IS NULL OR NOT EXISTS (
662                 SELECT 1
663                 FROM bookings b
664                 WHERE b.room_id = r.room_id
665                 AND b.status IN ('pending', 'confirmed') -- Exclude cancelled bookings
666                 AND (p_check_in_date < b.check_out_date AND p_check_out_date > b.check_in_date)
667             )
668         )
669     ORDER BY r.price ASC; -- Sort by price
670 END;
671 $$ LANGUAGE plpgsql;
672
673
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 70 msec.
```

For Host:

1.cancel_bookings_by_room: Creates a function that lets hosts batch-cancel all active bookings for a given room by invoking the cancel_booking helper to update status, log cancellations with refunds, and remove related reviews.

```
--HOST--
-- 1) Helper: cancel all bookings for a given room_id
CREATE OR REPLACE FUNCTION cancel_bookings_by_room(
    p_room_id  INTEGER,
    p_reason   TEXT
) RETURNS VOID LANGUAGE plpgsql AS $$
DECLARE
    rec RECORD;
BEGIN
    FOR rec IN
      SELECT booking_id
        FROM bookings
       WHERE room_id = p_room_id
         AND status   <> 'cancelled'
    LOOP
      -- your existing cancel_booking() will update status,
      -- insert into cancellation, and delete the review
      PERFORM cancel_booking(rec.booking_id, p_reason);
    END LOOP;
END;
$$;
```

Data Output | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 57 msec.
```

Triggers:

For Admin:

1.before_delete_hotel_cleanup: Creates a trigger that, before a hotel is deleted, cancels and logs all its bookings, removes its amenity links, and deletes its rooms to prevent foreign-key errors.

```
1148  $$;
1149
1150  -- 3) trigger function: before deleting a hotel, clean up everything
1151  CREATE OR REPLACE FUNCTION before_delete_hotel_cleanup()
1152  RETURNS TRIGGER LANGUAGE plpgsql AS $$
1153  BEGIN
1154      -- 3a) cancel & log its bookings
1155      PERFORM cancel_bookings_by_hotel(OLD.hotel_id, 'Hotel deleted');
1156      -- 3b) drop its amenity links
1157      DELETE FROM hotel_amenity
1158       WHERE hotel_id = OLD.hotel_id;
1159      -- 3c) drop its rooms
1160      DELETE FROM rooms
1161       WHERE hotel_id = OLD.hotel_id;
1162
1163      -- now the hotel row itself can be removed without FK errors
1164      RETURN OLD;
1165  END;
1166  $$;
1167
1168  -- attach the trigger:
1169  DROP TRIGGER IF EXISTS trg_before_hotel_delete ON hotel;
1170  CREATE TRIGGER trg_before_hotel_delete
1171  BEFORE DELETE ON hotel
1172  FOR EACH ROW
1173  EXECUTE FUNCTION before_delete_hotel_cleanup();
1174
1175
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 50 msec.
```

2.trg_delete_user_associated_records: This trigger ensures that when a user is deleted from the `users` table, any associated records in the `guest` and `host` tables (with the same `user_id`) are also

deleted automatically. It's a cleanup operation to maintain data consistency and avoid orphan records.

```
1142
1143  -- 1. Create the trigger function
1144  CREATE OR REPLACE FUNCTION trg_delete_user_associated_records()
1145  RETURNS TRIGGER AS $$
1146  BEGIN
1147      -- Delete from guest table if exists
1148      DELETE FROM guest WHERE user_id = OLD.user_id;
1149
1150      -- Delete from host table if exists
1151      DELETE FROM host WHERE user_id = OLD.user_id;
1152
1153      RETURN OLD;
1154  END;
1155  $$ LANGUAGE plpgsql;
1156
1157  -- 2. Create the trigger
1158  DROP TRIGGER IF EXISTS trg_before_delete_user ON users;
1159  CREATE TRIGGER trg_before_delete_user
1160  BEFORE DELETE ON users
1161  FOR EACH ROW
1162  EXECUTE FUNCTION trg_delete_user_associated_records();
1163
```

For Guest:

1.trg_booking_cancel_cleanup: Creates a trigger that, when a booking's status is updated to "cancelled," automatically invokes the cancel_booking function to log the cancellation, issue refunds, and clean up related reviews.

```sql

-- A) When a booking's status flips to 'cancelled', run cancel_booking(...)
CREATE OR REPLACE FUNCTION trg_booking_cancel_cleanup()
RETURNS TRIGGER LANGUAGE plpgsql AS $$
BEGIN
  IF TG_OP = 'UPDATE'
     AND OLD.status <> 'cancelled'
     AND NEW.status = 'cancelled'
  THEN
    PERFORM cancel_booking(NEW.booking_id, 'Cancelled via app');
  END IF;
  RETURN NEW;
END;
$$;

DROP TRIGGER IF EXISTS trg_on_booking_update ON bookings;
CREATE TRIGGER trg_on_booking_update
  AFTER UPDATE OF status ON bookings
  FOR EACH ROW
  EXECUTE FUNCTION trg_booking_cancel_cleanup();


-- B) When a payment details row is inserted, mark its booking confirmed
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 60 msec.
```

2.trg_payment_confirm_booking: Creates a trigger that automatically marks a booking as “confirmed” by updating its status whenever a new payment record is inserted.

```
1374
1375
1376  -- B) When a payment_details row is inserted, mark its booking confirmed
1377  CREATE OR REPLACE FUNCTION trg_payment_confirm_booking()
1378  RETURNS TRIGGER LANGUAGE plpgsql AS $$
1379  BEGIN
1380    UPDATE bookings
1381       SET status = 'confirmed'
1382     WHERE booking_id = NEW.booking_id;
1383    RETURN NEW;
1384  END;
1385  $$;
1386
1387  DROP TRIGGER IF EXISTS trg_on_payment_insert ON payment_details;
1388  CREATE TRIGGER trg_on_payment_insert
1389    AFTER INSERT ON payment_details
1390    FOR EACH ROW
1391    EXECUTE FUNCTION trg_payment_confirm_booking();
1392
1393
1394
```

Data Output | Messages | Notifications

```
ERROR:  trigger "trg_on_payment_insert" for relation "payment_details" already exists

SQL state: 42710
```

3.trg_notify_booking_creation: Creates a trigger that, after a new booking is inserted, looks up the guest's user ID and inserts a notification message into the notifications table.

```
1517
1518  -- 1) Notify guest on new booking
1519  CREATE OR REPLACE FUNCTION trg_notify_booking_creation()
1520  RETURNS TRIGGER LANGUAGE plpgsql AS $$
1521  DECLARE
1522    v_user_id   INT;
1523    v_message   TEXT;
1524  BEGIN
1525    -- look up the guest's user_id
1526    SELECT g.user_id INTO v_user_id
1527      FROM guest g
1528     WHERE g.guest_id = NEW.guest_id;
1529
1530    -- craft a message
1531    v_message := FORMAT(
1532      'Your booking #%s at room %s has been created with status "%s".',
1533      NEW.booking_id,
1534      NEW.room_id,
1535      NEW.status
1536    );
1537
1538    -- insert into notifications
1539    INSERT INTO notifications (user_id, message)
1540    VALUES (v_user_id, v_message);
1541
1542    RETURN NEW;
1543  END;
1544  $$;
1545
1546  DROP TRIGGER IF EXISTS trg_after_booking_insert ON bookings;
1547  CREATE TRIGGER trg_after_booking_insert
1548    AFTER INSERT ON bookings
1549    FOR EACH ROW
1550    EXECUTE FUNCTION trg_notify_booking_creation();
1551
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 69 msec.
```

4.trg_notify_booking_cancellation: Creates a trigger that, when a booking's status changes to "cancelled," looks up the guest's user ID and inserts a cancellation notification with the reason into the notifications table.

```sql

-- 2) Notify guest on booking cancellation
CREATE OR REPLACE FUNCTION trg_notify_booking_cancellation()
RETURNS TRIGGER LANGUAGE plpgsql AS $$
DECLARE
  v_user_id INT;
  v_message TEXT;
BEGIN
  -- only fire when status flips to 'cancelled'
  IF TG_OP = 'UPDATE'
     AND OLD.status <> 'cancelled'
     AND NEW.status = 'cancelled'
  THEN
    -- guest user
    SELECT g.user_id INTO v_user_id
      FROM guest g
     WHERE g.guest_id = NEW.guest_id;

    v_message := FORMAT(
      'Your booking #%s has been cancelled. Reason: %s',
      NEW.booking_id,
      COALESCE(NEW.status_reason, 'No reason provided')
    );

    INSERT INTO notifications (user_id, message)
    VALUES (v_user_id, v_message);
  END IF;

  RETURN NEW;
END;
$$;

DROP TRIGGER IF EXISTS trg_after_booking_update_notify ON bookings;
CREATE TRIGGER trg_after_booking_update_notify
  AFTER UPDATE OF status ON bookings
  FOR EACH ROW
  EXECUTE FUNCTION trg_notify_booking_cancellation();

```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 55 msec.
```

For Host:

1.before_delete_room_cleanup: Creates a trigger that, before deleting a room, cancels and logs all its bookings, nullifies their room_id to avoid foreign-key errors, and then allows the room row to be removed.

```
1416
1417
1418  -- 2) Trigger function: before deleting a room, clean up its bookings
1419  CREATE OR REPLACE FUNCTION before_delete_room_cleanup()
1420  RETURNS TRIGGER LANGUAGE plpgsql AS $$
1421  BEGIN
1422      -- 2a) cancel & log all bookings for this room
1423      PERFORM cancel_bookings_by_room(OLD.room_id, 'Room deleted');
1424
1425      -- 2b) detach any remaining bookings so the FK check passes
1426      UPDATE bookings
1427         SET room_id = NULL
1428       WHERE room_id = OLD.room_id;
1429
1430      -- now Postgres can delete the room row itself
1431      RETURN OLD;
1432  END;
1433  $$;
1434
1435
1436  -- 3) Attach the trigger to rooms
1437  DROP TRIGGER IF EXISTS trg_before_room_delete ON rooms;
1438  CREATE TRIGGER trg_before_room_delete
1439    BEFORE DELETE ON rooms
1440    FOR EACH ROW
1441    EXECUTE FUNCTION before_delete_room_cleanup();
1442
1443
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 86 msec.
```

2.notify_host_on_hotel_deletion: Creates a trigger that sends a notification to the host when their hotel is deleted, informing them of the deletion.

```
 -------
CREATE OR REPLACE FUNCTION notify_host_on_hotel_deletion()
RETURNS TRIGGER AS $$
BEGIN
    -- Insert a notification for the host
    INSERT INTO notifications (user_id, message)
    SELECT
        u.user_id,
        'Your hotel "' || OLD.name || '" has been deleted.'
    FROM host h
    JOIN users u ON h.user_id = u.user_id
    WHERE h.host_id = OLD.host_id;

    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER hotel_deletion_notification
AFTER DELETE ON hotel
FOR EACH ROW
EXECUTE FUNCTION notify_host_on_hotel_deletion();

```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 118 msec.
```

Database Indexing:

```sql
CREATE INDEX idx_users_username ON users(user_name);

CREATE INDEX idx_users_email ON users(email);

CREATE INDEX idx_discounts_valid_until ON discounts(valid_until DESC);

CREATE INDEX idx_bookings_booking_date ON bookings(booking_date DESC);

CREATE INDEX idx_city_hotel_id ON hotel(city);

CREATE INDEX idx_discounts_promo_valid ON discounts(promo_code, valid_until);

CREATE INDEX idx_bookings_guest_checkin ON bookings(guest_id, check_in_date);

CREATE INDEX idx_bookings_guest_id ON bookings(guest_id);

CREATE INDEX idx_bookings_room_dates ON bookings(room_id, check_in_date, check_out_date);

CREATE INDEX idx_payment_details_date ON payment_details(payment_date DESC);

CREATE INDEX idx_payment_details_booking_id ON payment_details(booking_id);

CREATE INDEX idx_host_user_id ON host(user_id);

CREATE INDEX idx_review_booking_id ON review(booking_id);

CREATE INDEX idx_review_hotel_id ON review(hotel_id);
```

1. User Authentication & Registration:

1.1 Index on `users(user_name)`

Table: `users`

Columns: `user_name`

Index Type: B-Tree Index

Reasoning:

Accelerates username lookups during login.

Avoids full table scans for authentication.

Query Example:

SELECT user_id, password_hashed, role FROM users WHERE user_name = %s;

Creation Command:

CREATE INDEX idx_users_username ON users(user_name);

1.3 Index on `users(email)`

Table: `users`

Columns: `email`

Index Type: B-Tree Index

Reasoning:

Optimizes email uniqueness checks during registration.

Query Example:

SELECT user_id FROM users WHERE email = %s;

Creation Command:

CREATE INDEX idx_users_email ON users(email);

---

2. Admin Dashboard

2.1 Index on `discounts(valid_until DESC)`

Table: `discounts`

Columns: m

Index Type: B-Tree Index (Descending)

Reasoning:

Optimizes filtering active discounts and sorting by expiry.

Query Example:

SELECT * FROM discounts WHERE valid_until >= CURRENT_DATE ORDER BY valid_until DESC;

Creation Command:

CREATE INDEX idx_discounts_valid_until ON discounts(valid_until DESC);

2.2 Index on `bookings(booking_date DESC)`

Table: `bookings`

Columns: `booking_date`

Index Type: B-Tree Index (Descending)

Reasoning:

Speeds up sorting for the "recent bookings" view.

Query Example:

SELECT * FROM v_recent_bookings ORDER BY booking_date DESC;

Creation Command:

CREATE INDEX idx_bookings_booking_date ON bookings(booking_date DESC);

2.3Index on `discounts(promo_code,valid_until)`

Table: `discounts`

Columns: `promo_code`

Index Type: B-Tree Index

Reasoning:

Enables instant promo code validation.

Query Example:

SELECT discount_amount FROM discounts WHERE promo_code = %s AND valid_until >= CURRENT_DATE;

Creation Command:

CREATE INDEX idx_discounts_promo_valid ON discounts(promo_code, valid_until);

---

3. Guest Dashboard

3.1Composite Index on `bookings(guest_id, check_in_date)`

Table: `bookings`

Columns: `guest_id`, `check_in_date`

Index Type: Composite B-Tree Index

Reasoning:

Efficiently filters and sorts upcoming stays for a guest.

Query Example:

SELECT * FROM bookings WHERE guest_id = %s AND check_in_date > CURRENT_DATE ORDER BY check_in_date;

Creation Command:

CREATE INDEX idx_bookings_guest_checkin ON bookings(guest_id, check_in_date);

3.2 Index on `bookings(guest_id)`

Table: `bookings`

Columns: `guest_id`

Index Type: B-Tree Index

Reasoning:

Speeds up guest-specific booking history retrieval.

Query Example:

SELECT * FROM bookings WHERE guest_id = %s;

Creation Command:

CREATE INDEX idx_bookings_guest_id ON bookings(guest_id);

---

4. Room Search & Booking

4.1 Composite Index on `bookings(room_id, check_in_date, check_out_date)`

Table: `bookings`

Columns: `room_id`, `check_in_date`, `check_out_date`

Index Type: Composite B-Tree Index

Reasoning:

Critical for fast overlap checks during room availability validation.

Query Example:

SELECT NOT EXISTS (

```
  SELECT 1 FROM bookings
  WHERE room_id = %s
  AND check_in_date < %s
  AND check_out_date > %s
);
```

Creation Command:

```
CREATE INDEX idx_bookings_room_dates ON bookings(room_id, check_in_date, check_out_date);
```

---

5. Payment & Booking History

5.1 Index on `payment_details(payment_date DESC)`

Table: `payment_details`

Columns: `payment_date`

Index Type: B-Tree Index (Descending)

Reasoning:

Speeds up sorting payment history by date.

Query Example:

```
SELECT pd.* FROM payment_details pd
JOIN bookings b ON pd.booking_id = b.booking_id
WHERE b.guest_id = %s
ORDER BY pd.payment_date DESC;
```

Creation Command:

CREATE INDEX idx_payment_details_date ON payment_details(payment_date DESC);

5.2 Index on `payment_details(`booking_id`)`

Table: `payment_details`

Columns: `payment_date`

Index Type: B-Tree Index (Descending)

Reasoning:

When PostgreSQL executes the join, it needs to quickly find all rows in `payment_details` that match a given `booking_id` from the `bookings` table.

Having an index on `booking_id` allows the planner to use a nested loop join or index lookup, rather than doing a slower sequential scan on the `payment_details` table.

Query Example:

SELECT pd.* FROM payment_details pd

JOIN bookings b ON pd.booking_id = b.booking_id

WHERE b.guest_id = %s

ORDER BY pd.payment_date DESC;

Creation Command:

CREATE INDEX idx_payment_details_booking_id ON payment_details(booking_id);

---

6. Host Dashboard

6.1 Index on `host(user_id)`

Table: `host`

Columns: `user_id`

Index Type: B-Tree Index

Reasoning:

Quickly retrieves host-owned hotels.

Query Example:

```
SELECT h.* FROM hotel h
JOIN host ho ON h.host_id = ho.host_id
WHERE ho.user_id = %s;
```

Creation Command:

```
CREATE INDEX idx_host_user_id ON host(user_id);
```

---

7. Reviews

7.1 Index on `review(booking_id)`

Table: `review`

Columns: `booking_id`

Index Type: B-Tree Index

Reasoning:

Ensures fast lookups for existing reviews on a booking.

Query Example:

INSERT INTO review (rating, comment, review_date, booking_id, guest_id, hotel_id)

VALUES

(4.0, 'Very comfortable', '2025-04-13 10:00:00', 2, 2, 1);

Creation Command:

CREATE INDEX idx_review_booking_id ON review(booking_id);

7.2 Index on `review(hotel_id)`

Table: `review`

Columns: `hotel_id`

Index Type: B-Tree Index

Reasoning:

Aggregates reviews efficiently for hotel rating calculations.

Query Example:

SELECT AVG(rating) FROM review WHERE hotel_id = %s;

Creation Command:

CREATE INDEX idx_review_hotel_id ON review(hotel_id);

All Roles and list of privileges given to them:

This mapping ensures a strict separation of duties:

Admin = full control

Host = manage hotels & rooms only

Guest = view hotels/rooms + handle their own bookings/payments

1. `admin`

Tables & Views:

`GRANT ALL PRIVILEGES` on all existing tables in schema `public`

Default for future tables in `public`: also `ALL PRIVILEGES`

Sequences:

`GRANT ALL PRIVILEGES` on all existing sequences in `public`

Default for future sequences in `public`: also `ALL PRIVILEGES`

Functions/Procedures:

`GRANT EXECUTE` on all functions in schema `public`

`GRANT EXECUTE` on all pgcrypto functions (e.g. `gen_random_uuid()`, `crypt()`)

Extensions:

GRANT USAGE on schema pgcrypto

2. host

Tables

GRANT SELECT, INSERT, UPDATE, DELETE on just the hotel and rooms tables

Sequences

GRANT USAGE, SELECT on hotel_hotel_id_seq and rooms_room_id_seq

3. guest:

Tables:

GRANT SELECT on hotel and rooms

GRANT SELECT, INSERT, UPDATE on bookings and payment_details

GRANT INSERT on login_attempts

Sequences:

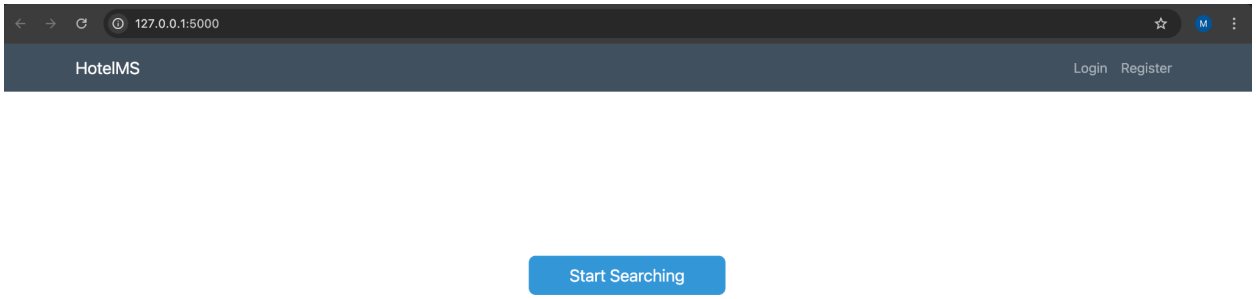
GRANT USAGE, SELECT on bookings_booking_id_seq and payment_details_payment_id_seq

```sql
-- Create roles
CREATE ROLE guest;
ALTER ROLE guest WITH PASSWORD 'guest';
CREATE ROLE host;
CREATE ROLE admin;


-- Grant privileges to `hotel_app` user
-- GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO hotel_app
-- GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public TO hotel_app;


-- Tables/Views
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO admin;

-- Sequences
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO admin;

-- Functions/Procedures
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA public TO admin;

-- Extensions (e.g., pgcrypto)
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA pgcrypto TO admin;
-- For future tables/views
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT ALL PRIVILEGES ON TABLES TO admin;

-- For future sequences
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT ALL PRIVILEGES ON SEQUENCES TO admin;
```

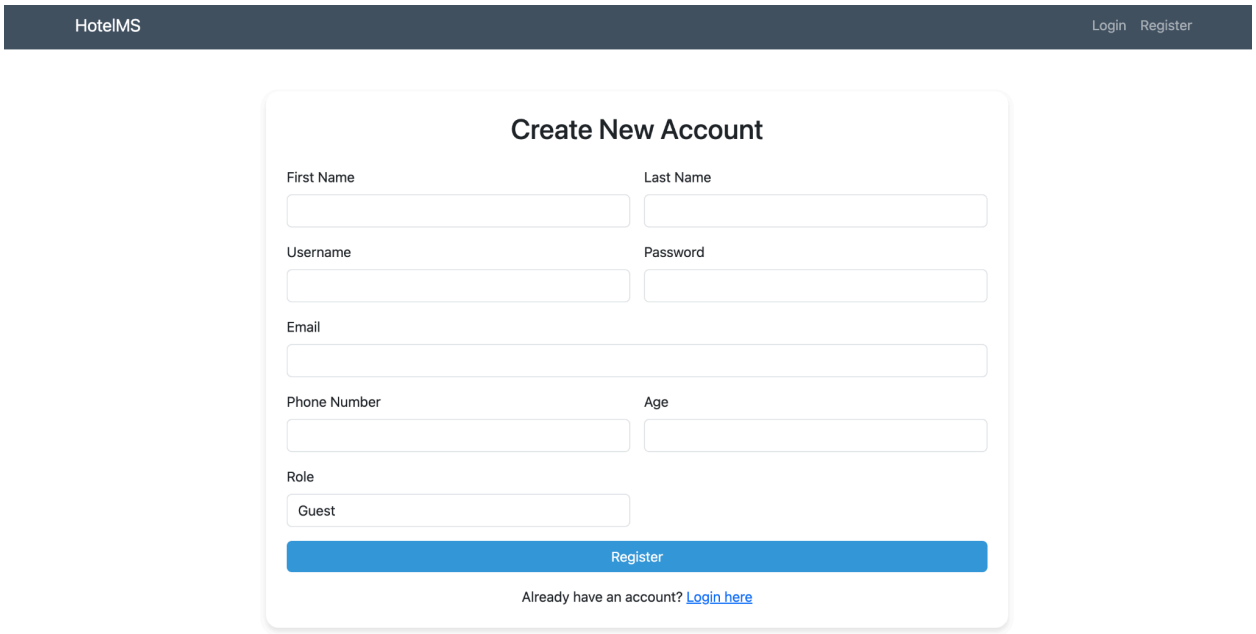
```sql
GRANT ALL PRIVILEGES ON TABLES TO admin;

-- For future sequences
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT ALL PRIVILEGES ON SEQUENCES TO admin;

-- For future functions
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT EXECUTE ON FUNCTIONS TO admin;
GRANT SELECT, INSERT, UPDATE, DELETE ON hotel, rooms TO admin;

-- For pgcrypto functions like gen_random_uuid(), crypt(), etc.
GRANT USAGE ON SCHEMA pgcrypto TO admin;
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA pgcrypto TO admin;

-- Grant privileges to `host`
GRANT SELECT, INSERT, UPDATE, DELETE ON hotel, rooms TO host;
GRANT USAGE, SELECT ON SEQUENCE hotel_hotel_id_seq, rooms_room_id_seq TO host;

-- Grant privileges to `guest`
GRANT SELECT ON hotel, rooms TO guest;
GRANT SELECT, INSERT, UPDATE ON bookings, payment_details TO guest;
GRANT USAGE, SELECT ON SEQUENCE bookings_booking_id_seq, payment_details_payment_id_seq TO guest;
GRANT INSERT ON login_attempts To guest;
-- Verify privileges for `host`
SELECT table_name, privilege_type
FROM information_schema.table_privileges
WHERE grantee = 'host';


---
```

FRONTEND

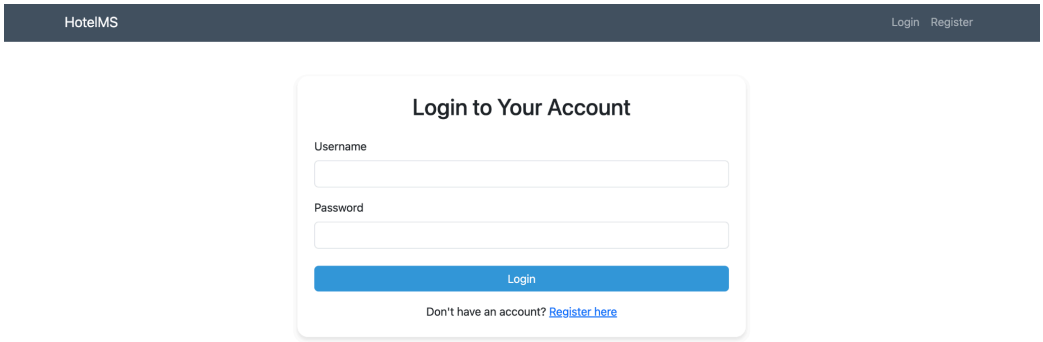
Search page :



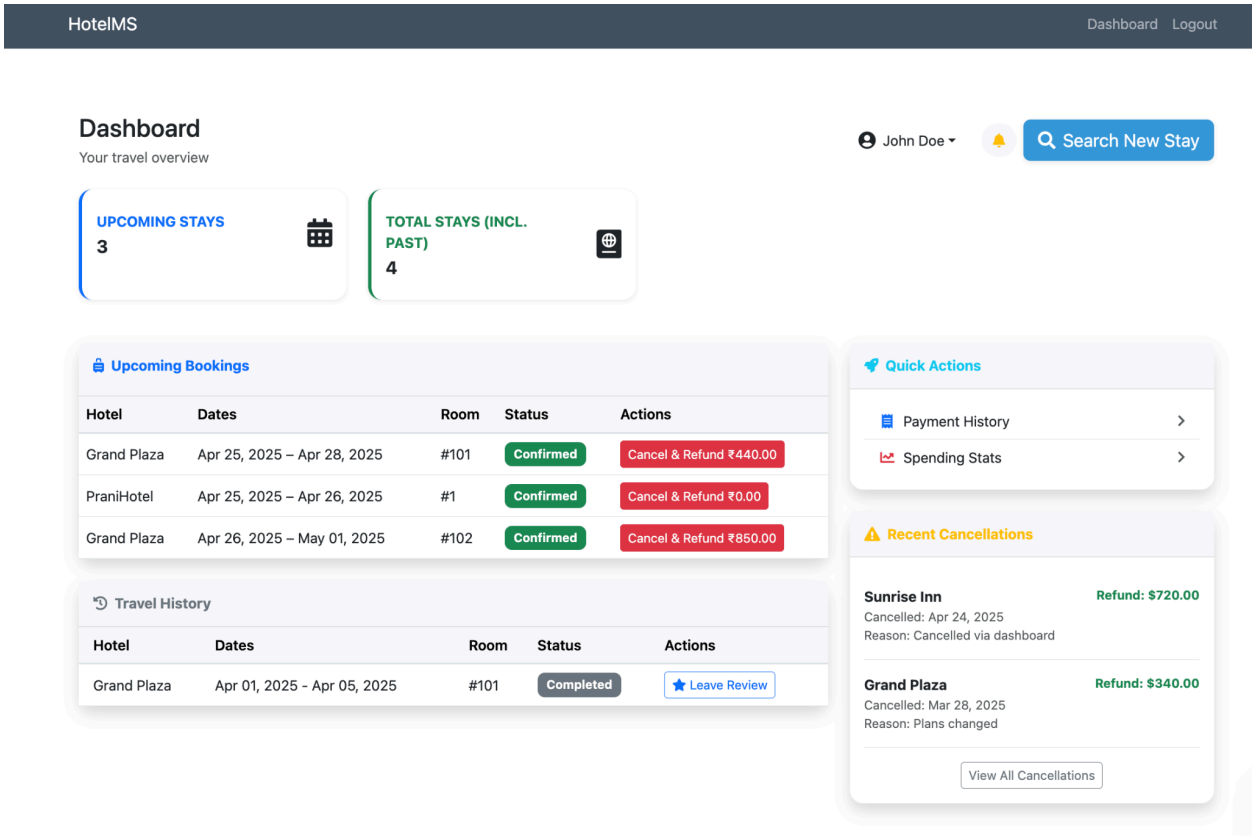
Registering page :



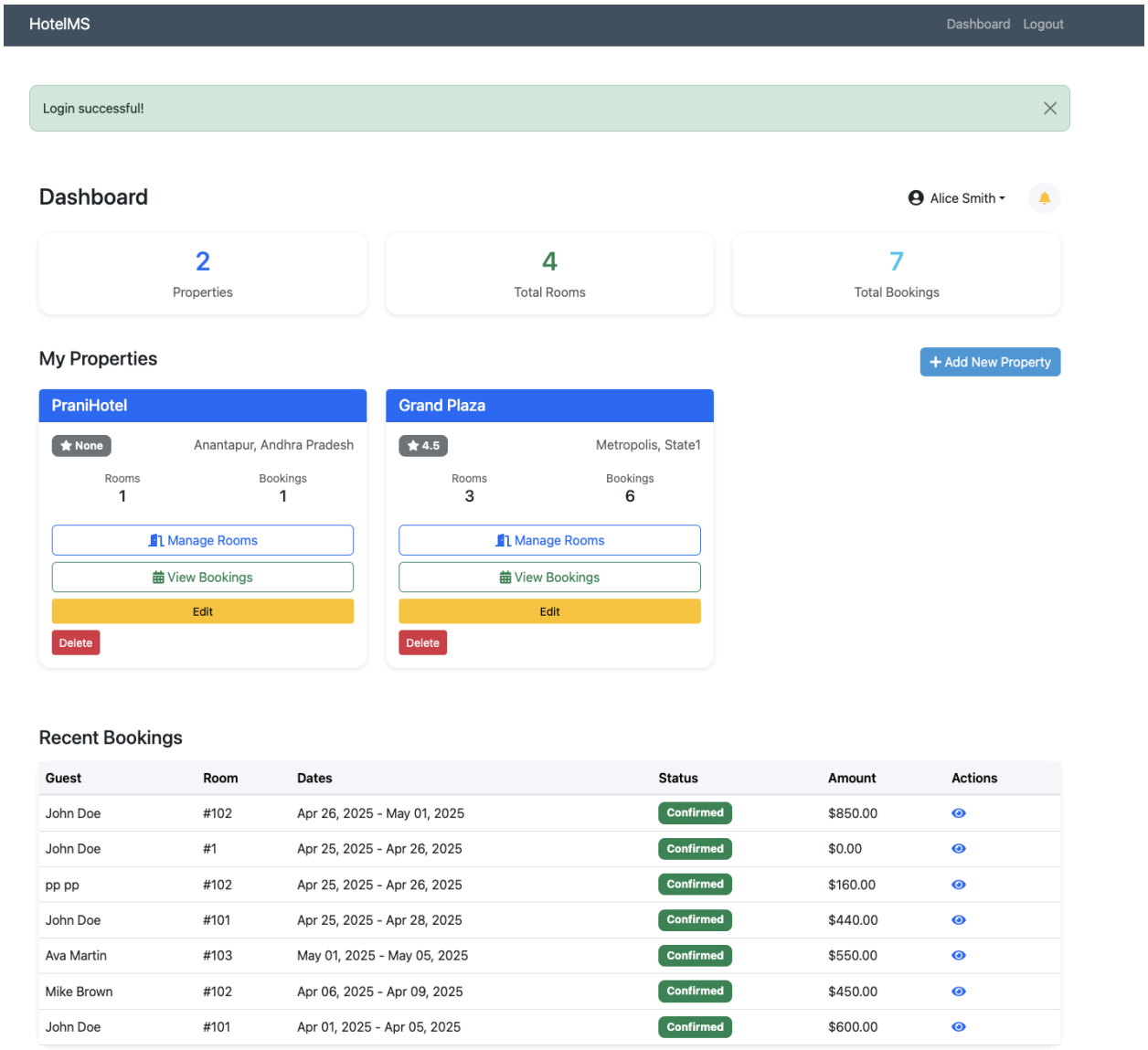
Login page :



Guest dashboard:



Host dashboard:



Admin database:

Admin Dashboard

Send Notification to Users

Manage Discounts

Promo Code	Description	Discount Amount	Valid From	Valid Until
SUMMER25	Seasonal discount	$ 25.00	dd/mm/yyyy	dd/mm/yyyy

Create Discount

Total Users	Total Bookings	Total Hotels
14	14	5

Recent Login Attempts

Timestamp	Username	Status
25 Apr 2025 02:17	admin	Successful
25 Apr 2025 02:17	user2	Successful
25 Apr 2025 01:44	user1	Successful
25 Apr 2025 01:44	pp	Successful
25 Apr 2025 01:43	user1	Successful
25 Apr 2025 01:41	user1	Successful
25 Apr 2025 01:30	user1	Successful
25 Apr 2025 01:29	admin	Successful
25 Apr 2025 01:23	user2	Successful
25 Apr 2025 01:23	user1	Successful

Recent Bookings (Last 5)

Booking ID	User	Hotel	Dates	Status	Amount
#16		Grand Plaza	26 Apr 01 May	Confirmed	$850.00
#15		PraniHotel	25 Apr 26 Apr	Confirmed	$0.00
#14		Grand Plaza	25 Apr 26 Apr	Confirmed	$160.00
#13		Grand Plaza	25 Apr 28 Apr	Confirmed	$440.00
#10		Mountain Retreat	16 May 20 May	Confirmed	$910.00

Recent Users (Last 5)

User ID	Username	Email	Role	Joined
#22	ppp	ppp@gmail.com	Guest	24 Apr 2025 16:00
#20	pp	pp@gmail.com	Guest	24 Apr 2025 16:00
#12	admin	None	Admin	24 Apr 2025 15:17
#10	user10	user10@example.com	Guest	01 Mar 2025 08:45
#9	user9	user9@example.com	Guest	01 Mar 2025 08:40