Luxury Real Estate Platform

1. Project Overview

Build a luxury real estate platform with **modern UI/UX** (glassmorphic cards, 3D property previews, parallax scrolling). Backend manages users, properties, bookings, and payments. Support Stripe and bKash. Include **advanced algorithms, OOP, and data structures**.

2. Requirements

2.1 Functional Requirements

2.1.1 User Management

- Users can register and log in.
- User data stored in `Users` table (PostgreSQL).
- Email must be unique.
- Users can view booked property visits and payment history.

2.1.2 Property Management

- Admin can create, update, delete properties.
- Property fields: `id (PK), name, slug (unique), description, location, price, bedrooms, bathrooms, amenities, status (active/inactive), timestamps`
- Users can view property lists and details.
- Include hierarchical categories (residential, commercial, etc.)

2.1.3 Booking/Inquiry Management

- Bookings belong to users.
- Fields:
  `id, user_id (FK → Users.id), property_id (FK), total_amount, status (pending, paid, canceled), timestamps`

2.1.4 Payment System

- Stripe and bKash support
- Stripe:
  - Create payment intent, confirm, webhook
- bKash:
  - Checkout, execute, query
- Payment table:
  `id, booking_id (FK), provider, transaction_id, status, raw_response, timestamps`

2.1.5 Booking Flow

- User selects property → creates booking.
- User chooses the payment provider.
- System initiates payment → confirms/fails → updates booking.
- Ensure property availability management.

2.2 Core Design & Algorithm Requirements

2.2.1 OOP Requirement

- Classes:
  - `User` – registration, login, booking history
  - `Property` – CRUD operations, availability
  - `Booking` – calculate total/subtotal, update status
  - `Payment` – strategy pattern for Stripe/bKash

2.2.2 Data Structure Requirement

- PostgreSQL tables + MongoDB for caching or media assets.
- Indexed fields and hierarchical category tree.

2.2.3 Algorithm Requirement

- Booking totals/subtotals
- Availability checking with safe concurrency
- DFS for property recommendation in category tree

2.2.4 Design Pattern Requirement

- Strategy pattern for Payment System

2.2.5 DFS + Caching Requirement

- DFS traversal of property categories for recommendations.
- Cache category tree in Redis/memcached.

3. Non-Functional Requirements

- Clean REST API
- Django migrations
- Proper data validation
- Secure API keys
- Logging & error handling
- Scalable for adding more payment providers

4. Deliverables

4.1 Documentation

- System architecture diagram
- ERD (Users, Properties, Bookings, Payments)
- API documentation (Postman/Swagger)
- Payment flow diagrams

4.2 Code Deliverables

- Backend: Django REST Framework
- Frontend: Next.js + React + Three.js + TailwindCSS
- Seeders for admin and sample properties

4.3 Payment Integrations

- Stripe (test + live)
- bKash (sandbox + live)
- Webhook handlers

4.4 Testing

- Unit tests for models
- API tests for authentication, bookings, payments