MISRIMAL NAVAJEE MUNOTH JAIN ENGINEERINGCOLLEGE

(Managed By Tamil Nadu Educational and MedicalTrust) Thoraipakkam, Chennai – 600097.

NM1042

MERN Stack Powered by MongoDB

(Urban Nest)

REGULATION – 2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NAME :

REGISTER NUMBER :

YEAR : IV

SEMESTER : VII

MISRIMAL NAVAJEE MUNOTH JAIN ENGINEERING COLLEGE

(Managed By Tamil Nadu Educational and Medical Trust)Thoraipakkam, Chennai – 600097.

DEPARTMENT OF COMPUTER SCIENCE ANDENGINEERING

Register Number

BONAFIDE CERTIFICATE

This is to certify that this is a bonafide record of work done by_____of IV-Year **B.E-Computer Science and Engineering** in the **NM1042-MERN Stack powered by MongoDB Laboratory** during the Academic year **2024-2025**

Staff In-Charge. **Head of the Department**

Submitted for the University Practical Examination held on

Internal Examiner **External Examiner**

MISRIMAL NAVAJEE MUNOTH JAIN ENGINEERING COLLEGE, CHENNAI – 97

DEPARTMENT OFCOMPUTER SCIENCE AND ENGINEERING

VISION
Producing competent Computer Engineers with a strong background in the latest trendsand technology to achieve academic excellence and to become pioneer in software and hardware products with an ethical approach to serve the society
MISSION
To provide quality education in Computer Science and Engineering with the state of the art facilities. To provide the learning audience that helps the students to enhance problem solving skills and to inculcate in them the habit of continuous learning in their domain of interest. To serve the society by providing insight solutions to the real world problems by employing the latest trends of computing technology with strict adherence to professional and ethical responsibilities.

Project Overview

Purpose

The *Urban Nest* is a comprehensive platform designed to simplify the process of renting properties, bridging the gap between property owners and renters. The app aims to streamline the rental journey, from property browsing to finalizing lease agreements. Its purpose is to:

1. **Simplify Property Rentals**: Provide an easy-to-use interface for renters to search for properties, filter options, and book rentals seamlessly.
2. **Enhance Owner-Renter Communication**: Enable direct and secure communication between property owners and prospective tenants.
3. **Centralize Rental Operations**: Offer property owners a platform to list, manage, and track their rental properties efficiently.
4. **Ensure Platform Governance**: Allow admins to maintain a secure, trustworthy, and well-regulated environment for all users.
5. **Promote Scalability and Efficiency**: Utilize MERN stack technologies to deliver a responsive, robust, and scalable application suitable for high traffic and diverse user needs.

Features

The *Urban Nest* is equipped with a variety of features that cater to renters, property owners, and administrators:

1. Renter Features:

- **User Registration and Login**: Renters can create accounts securely with their email and password.
- **Property Browsing and Search Filters**: Renters can explore a catalog of properties and use filters like location, rent range, property type, and amenities to find suitable options.
- **Property Details and Inquiry**: Each property includes detailed descriptions, images, and owner contact information. Renters can send inquiries through a small form.
- **Booking Management**: Renters can view their bookings with real-time status updates such as ”Pending” or ”Confirmed.”

2. Owner Features:

- **Admin Approval for Owner Accounts**: Only verified users can register as property owners.
- **Property Management**: Owners can add, update, delete, and manage the status of their listed properties.
- **Booking Response**: Owners can approve or deny booking requests, updating renters about their status through the app.

3. Admin Features:

- **User Approval**: Admins can approve owner registrations after verifying their legitimacy.
- **Platform Governance**: Admins monitor activities and enforce platform policies, ensuring compliance with terms of service and privacy regulations.
- **Data and Activity Monitoring**: Admins maintain system integrity by tracking user actions and moderating content.

4. Technical Features:

- **Real-time Data Handling**: Powered by MongoDB, the app ensures quick and efficient data access and updates.
- **Secure Authentication**: JWT and bcrypt.js provide secure user authentication and data protection.
- **Responsive UI**: Built with React, Material UI, and Bootstrap, the frontend delivers an intuitive and visually appealing user experience.
- **RESTful APIs**: The Express.js backend provides seamless interaction between the client and server.
- **Media and File Management**: Multer enables smooth handling of property images and other files.

5. Additional Highlights:

- **Transaction and Lease Management**: Renters and owners can negotiate and finalize lease agreements securely within the app.
- **Notifications**: Users receive updates about booking confirmations and admin approvals.
- **Cross-Browser Compatibility**: The app is optimized for multiple browsers to provide a consistent user experience.

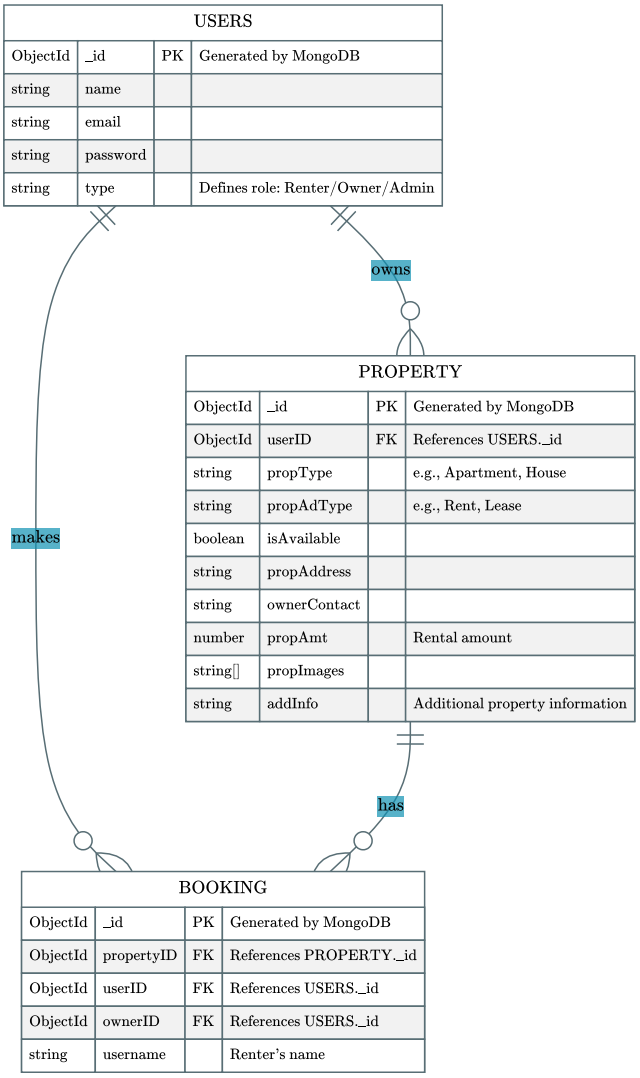
Technical Implementation

The *Urban Nest* leverages the MERN stack, utilizing MongoDB for database management, Express.js for backend logic, React.js for a dynamic frontend, and Node.js as the runtime environment. Key technologies and libraries include Moment.js for date handling, Axios for API communication, Ant Design for UI components, and Mongoose for database schema modeling.

The architecture is designed for scalability, real-time updates, and user-friendliness, ensuring a seamless rental experience for all stakeholders.

Architecture

MongoDB Schema



1. Frontend Architecture: Using React

The frontend of the Urban Nest is built using **React.js**, a powerful library for creating dynamic and responsive user interfaces. It follows a component-based architecture to ensure reusability and maintainability.

Key Components and Features:

- **Component Hierarchy**:
  - **App Component**: Acts as the root component and manages the routing for the entire application.
  - **Dashboard Component**: Displays a list of rental properties and provides filtering options.
  - **PropertyDetails Component**: Shows detailed information about a selected property.
  - **UserProfile Component**: Manages user-specific information such as bookings.

    - **AdminPanel Component**: Handles admin-specific actions like approving users and properties.
    - **OwnerPanel Component**: Allows property owners to manage their listings.
- **Routing**: React Router is used to manage routes for different user roles (Renter, Owner, Admin) and app features, ensuring smooth navigation.
- **UI Libraries**:
    - **Material UI** and **Ant Design (Antd)**: Enhance the UI with pre-built, responsive components.
    - **Bootstrap**: Provides additional styling flexibility.
- **State Management**:
    - **Context API** or **Redux** (optional): Used to manage global states such as user authentication and property data.
- **API Communication**:
    - **Axios**: Handles API requests to interact with the backend, fetching or submitting data efficiently.

2. Backend Architecture: Using Node.js and Express.js

The backend is developed with **Node.js** and the **Express.js framework** to handle server-side logic, routing, and API endpoints.

Key Features and Implementation:

- **RESTful API**: The backend exposes RESTful endpoints for CRUD operations on users, properties, and bookings.
- **Authentication and Authorization**:
    - **JWT (JSON Web Tokens)**: Ensures secure authentication for Renter, Owner, and Admin roles.
    - Middleware such as `authMiddleware.js` validates user sessions and role-based access.
- **Middleware**:
    - **CORS**: Allows cross-origin requests for the frontend.
    - **Body-parser**: Parses incoming request bodies.
    - **Multer**: Manages file uploads (e.g., property images).
- **Environmental Configurations**: Environment variables are stored in a `.env` file to manage sensitive data like database connection strings and JWT keys.
- **Libraries Used**:
    - **bcryptjs**: Secures user passwords by hashing them.
    - **dotenv**: Manages environment variables.
    - **nodemon**: Automatically restarts the server during development.

3. Database Architecture: Using MongoDB

The database uses **MongoDB**, a NoSQL database, to store and manage app data in a flexible, JSON-like format.

Database Collections:

1. **Users Collection**: Stores information about Renters, Owners, and Admins.

```
{
    "_id": "ObjectId",
    "name": "Alice",
    "email": "alice@example.com",
    "password": "hashed_password",
    "type": "Renter" // or "Owner" or "Admin"
}
```

2. **Properties Collection**: Stores rental property details added by Owners.

```
{
    "_id": "ObjectId",
    "userId": "OwnerId",
    "propType": "Apartment",
    "propAdType": "Rent",
    "isAvailable": true,
    "address": "123 Main St",
    "ownerContact": "123-456-7890",
    "propAmt": 1500,
    "propImages": ["image1.jpg", "image2.jpg"],
    "addInfo": "Near the park"
}
```

3. **Bookings Collection**: Tracks booking requests made by Renters.

```
{
    "_id": "ObjectId",
    "propertyId": "PropertyId",
    "userId": "RenterId",
    "ownerId": "OwnerId",
    "username": "Alice"
}
```

Database Interactions:

- **Mongoose ODM**:
  - Simplifies database operations like creating schemas, validations, and managing relations between collections.
  - Example:
    - `userId` in the **Properties** collection acts as a foreign key to the **Users** collection.
    - `propertyId` in the **Bookings** collection links to the **Properties** collection.

- **CRUD Operations**:
  - Create: Add new users, properties, and bookings.
  - Read: Fetch property listings, user details, or booking history.
  - Update: Modify property availability or booking status.
  - Delete: Remove users, properties, or bookings.

By combining React.js on the frontend, Node.js with Express.js on the backend, and MongoDB for data storage, this MERN-based architecture provides a robust and scalable solution for a Urban Nest.

Setup Instructions

To set up and run the **Urban Nest** locally, follow the steps below:

---

1. Prerequisites

Ensure you have the following installed on your system:

Software Dependencies:

- **Node.js and npm**:
  Install the latest version of Node.js, which includes npm.

  ```
  node -v
  npm -v
  ```

  Ensure these commands return installed versions.

- **MongoDB**:
  Install MongoDB Community Edition. You can either set it up locally or use MongoDB Atlas.
- **Git**:
  Install Git to clone the project repository.
  Download Git.

---

2. Clone the Repository

To download the project files:

```
git clone https://github.com/VSuryaPrashanth/UrbanNest.git
cd Rentify
```

---

3. Client (Frontend) Setup

Steps:

1. Navigate to the frontend directory:

   ```
   cd client
   ```

2. Install dependencies:

   ```
   npm install
   ```

3. Configure EmailJS for Quote Emailing (Optional):
   If you want to use EmailJS for emailing quotes, set up the following variables in a `.env` file inside the `client` directory. These variables can be obtained from your EmailJS account.

```
// .env
VITE_EMAILJS_SERVICE_NAME=<Your EmailJS Service Name>
VITE_EMAILJS_TEMPLATE_NAME=<Your EmailJS Template Name>
VITE_EMAILJS_PUBLIC_KEY=<Your EmailJS Public Key>
```

4. Update proxy settings (Optional):
   To use a local server during development, modify `vite.config.js` in the `client` directory:

```
export default defineConfig({
    server: {
        proxy: {
            '/api': 'http://localhost:5000/',
        },
    },
    plugins: [react()],
});
```

---

4. Server (Backend) Setup

Steps:

1. Navigate to the backend directory:

```
cd server
```

2. Install dependencies:

```
npm install
```

3. Configure environment variables:
   Create a `.env` file in the `server` directory based on the `.env_sample` file. Include the following values:

```
// .env
MONGO_URI=<Your MongoDB Connection String>
CLOUDINARY_CLOUD_NAME=<Your Cloudinary Cloud Name>
CLOUDINARY_API_KEY=<Your Cloudinary API Key>
CLOUDINARY_API_SECRET=<Your Cloudinary API Secret>
PORT=5000
JWT_SECRET=<Your JWT Secret>
```

- **MONGO_URI**: Connection string for your MongoDB database (local or Atlas).
- **Cloudinary Credentials**: For image uploads.
- **JWT_SECRET**: A secret key for JSON Web Token (JWT) authentication.

---

5. Run the Application Locally

Steps:

1. **Start the Backend Server**:
   Navigate to the backend directory and start the server:

   ```
   cd server
   npm start
   ```

   The backend will run at `http://localhost:5000`.

2. **Start the Frontend**:
   Navigate to the client directory and start the frontend:

   ```
   cd client
   npm start
   ```

   The frontend will run at `http://localhost:3000`.

---

6. Verify and Test

- Open your browser and navigate to `http://localhost:3000`.
- Test the features like property browsing, user registration, and booking functionalities.

Folder Structure

To ensure maintainability and scalability of the application, the folder structure for both the **frontend** (React) and **backend** (Node.js) is organized as follows:

---

1. Client: Structure of the React Frontend

The React frontend is structured to manage components, assets, and utilities effectively.

```
client
├── .env_sample
├── .eslintrc.cjs
├── .gitignore
├── dist
│   ├── assets
│   │   ├── defaultPP-D0m93DkQ.png
│   │   ├── index-BIKiTg-q.js
│   │   ├── index-DY9-Svk2.css
│   │   ├── logo-DBeKnw1a.png
│   │   └── welcome-BOjCqAiG.jpg
│   ├── index.html
│   ├── logo.png
│   ├── logo_1.png
│   └── vite.svg
├── node_modules
│   ├── .bin
│   ├── @alloc
│   :
├── index.html
├── package-lock.json
├── package.json
├── postcss.config.js
├── public
│   ├── logo.png
│   ├── logo_1.png
│   └── vite.svg
├── README.md
├── src
│   ├── api
│   │   ├── auth.js
│   │   └── property.js
│   ├── App.jsx
│   ├── AppContext.jsx
│   ├── assets
│   │   ├── defaultPP.png
│   │   ├── logo.png
│   │   ├── logofull.png
│   │   ├── react.svg
```

```
│   │   └── welcome.jpg
│   ├── components
│   │   ├── AddProperty.jsx
│   │   ├── CustomSwitch.jsx
│   │   ├── CustomSwitchBig.jsx
│   │   ├── EditProperty.jsx
│   │   ├── Find.jsx
│   │   ├── Header.jsx
│   │   ├── Liked.jsx
│   │   ├── Loader.jsx
│   │   ├── LoaderFull.jsx
│   │   ├── Login.jsx
│   │   ├── PropertyCard.jsx
│   │   ├── Rent.jsx
│   │   └── Signup.jsx
│   ├── hooks
│   │   └── useOutsideClick.js
│   ├── index.css
│   ├── main.jsx
│   └── pages
│       ├── Dashboard.jsx
│       └── Welcome.jsx
├── tailwind.config.js
├── vercel.json
└── vite.config.js
```

---

2. Server: Organization of the Node.js Backend

The backend is structured to keep models, routes, controllers, and middleware separated, ensuring modularity and ease of testing.

```
.
├── .env_sample
├── .gitignore
├── controllers
│   ├── propertyController.js
│   └── userController.js
├── node_modules
│   ├── .bin
│   ├── @alloc
│   :
├── index.js
├── middlewares
│   ├── authHandler.js
│   └── errorHandler.js
├── models
│   ├── propertyModel.js
│   └── userModel.js
├── package-lock.json
├── package.json
```

```
├── routes
│   ├── propertyRoutes.js
│   └── userRoutes.js
├── utils
│   ├── connectDB.js
│   └── imageUpload.js
└── vercel.json
```

Key Features of Folder Structure:

1. **Modularity**:
   - Separation of concerns between authentication, property management, and booking management.
   - Each module (e.g., **users**, **properties**, **bookings**) has its own model, routes, and controller.
2. **Scalability**:
   - As the application grows, new features can be added by creating new modules within the **controllers**, **models**, and **routes** folders.
3. **Code Reusability**:
   - Common components and utilities in the **frontend** (e.g., `Navbar`, `PropertyCard`) and **backend** (e.g., `authMiddleware`, `errorHandler`) are reused across the app.
4. **Ease of Deployment**:
   - Frontend and backend are separated, making it easy to deploy on platforms like Netlify (frontend) and Heroku or AWS (backend).

By adhering to this structure, development, testing, and maintenance of the **House Rent App** remain efficient and organized.

API Documentation for Urban Nest

Base URL:

```
http://localhost:5000/api
```

---

1. User Endpoints

1.1 User Registration

- **Endpoint:** `/users/register`
- **Method:** `POST`
- **Description:** Register a new user as a renter or owner.
- **Request Body:**

```
{
  "name": "Alice",
  "email": "alice@example.com",
  "password": "securepassword",
  "type": "renter" // or "owner"
}
```

- **Response:**
  - **Success:** `201 Created`

```
{
  "message": "User registered successfully",
  "user": {
    "_id": "user_id",
    "name": "Alice",
    "email": "alice@example.com",
    "type": "renter"
  }
}
```

  - **Failure:** `400 Bad Request`

```
{ "message": "Email already exists" }
```

1.2 User Login

- **Endpoint:** `/users/login`
- **Method:** `POST`

- **Description:** Log in an existing user.
- **Request Body:**

```
{
  "email": "alice@example.com",
  "password": "securepassword"
}
```

- **Response:**
  - **Success:** `200 OK`

```
{
  "message": "Login successful",
  "token": "jwt_token",
  "user": {
    "_id": "user_id",
    "name": "Alice",
    "email": "alice@example.com",
    "type": "renter"
  }
}
```

  - **Failure:** `401 Unauthorized`

```
{ "message": "Invalid credentials" }
```

---

2. Property Endpoints

2.1 Add Property

- **Endpoint:** `/properties/add`
- **Method:** `POST`
- **Description:** Add a new property (Owner only).
- **Headers:**

```
{
  "Authorization": "Bearer jwt_token"
}
```

- **Request Body:**

```
{
  "type": "apartment",
  "adType": "rent",
```

```
    "isAvailable": true,
    "address": "123 Main St, Cityville",
    "ownerContact": "1234567890",
    "amount": 1500,
    "images": ["image_url1", "image_url2"],
    "additionalInfo": "Spacious 2-bedroom apartment"
}
```

- **Response:**
  - **Success:** `201 Created`

```
{
  "message": "Property added successfully",
  "property": { /* property details */ }
}
```

  - **Failure:** `403 Forbidden`

```
{ "message": "Access denied" }
```

2.2 Get All Properties

- **Endpoint:** `/properties`
- **Method:** `GET`
- **Description:** Fetch all properties available for rent.
- **Response:**
  - **Success:** `200 OK`

```
[
  {
    "_id": "property_id",
    "type": "apartment",
    "adType": "rent",
    "isAvailable": true,
    "address": "123 Main St, Cityville",
    "ownerContact": "1234567890",
    "amount": 1500,
    "images": ["image_url1", "image_url2"],
    "additionalInfo": "Spacious 2-bedroom apartment"
  },
  ...
]
```

---

3. Booking Endpoints

3.1 Create Booking

- **Endpoint:** `/bookings/create`
- **Method:** `POST`
- **Description:** Create a new booking for a property.
- **Headers:**

```
{
  "Authorization": "Bearer jwt_token"
}
```

- **Request Body:**

```
{
  "propertyId": "property_id",
  "ownerId": "owner_id",
  "username": "Alice"
}
```

- **Response:**
  - **Success:** `201 Created`

```
{
  "message": "Booking created successfully",
  "booking": { /* booking details */ }
}
```

  - **Failure:** `400 Bad Request`

```
{ "message": "Property not available" }
```

3.2 Get User Bookings

- **Endpoint:** `/bookings/user`
- **Method:** `GET`
- **Description:** Fetch all bookings for the logged-in user.
- **Headers:**

```
{
  "Authorization": "Bearer jwt_token"
}
```

- **Response:**
  - **Success:** `200 OK`

```
[
  {
    "_id": "booking_id",
    "propertyId": "property_id",
    "ownerId": "owner_id",
    "username": "Alice",
    "status": "pending"
  },
   ...
]
```

---

4. Admin Endpoints

4.1 Approve Owner Account

- **Endpoint:** `/admin/approve-owner`
- **Method:** `PATCH`
- **Description:** Approve a user as an owner.
- **Headers:**

```
{
  "Authorization": "Bearer admin_jwt_token"
}
```

- **Request Body:**

```
{
  "userId": "user_id"
}
```

- **Response:**
  - **Success:** `200 OK`

```
{ "message": "Owner account approved successfully" }
```

  - **Failure:** `404 Not Found`

```
{ "message": "User not found" }
```

4.2 Monitor Activities

- **Endpoint:** `/admin/activity`
- **Method:** `GET`
- **Description:** Fetch recent activities (user registrations, bookings, etc.).
- **Headers:**

```
{
  "Authorization": "Bearer admin_jwt_token"
}
```

- **Response:**
  - **Success:** `200 OK`

```
[
  { /* activity details */ },
  ...
]
```

---

5. Miscellaneous

5.1 Get Property Details

- **Endpoint:** `/properties/:id`
- **Method:** `GET`
- **Description:** Fetch detailed information about a specific property.
- **Response:**
  - **Success:** `200 OK`

```
{ /* property details */ }
```

  - **Failure:** `404 Not Found`

```
{ "message": "Property not found" }
```

Authentication

In the **Urban Nest** developed using the MERN stack, authentication and authorization are implemented to ensure secure access and role-based permissions for different types of users: Renter, Owner, and Admin. Below is a detailed explanation of the techniques and tools used:

Authentication Mechanism

Authentication verifies the identity of a user before granting access to the system. The Urban Nest implements authentication using **JSON Web Tokens (JWT)**.

Steps Involved:

1. **User Registration:**
   - Users (Renters or Owners) register on the app by providing their email, name, password, and role (Renter/Owner).
   - The password is hashed using **bcryptjs** before being stored in the database for enhanced security.
2. **Login Process:**
   - Users log in by providing their email and password.
   - The provided password is verified against the hashed password stored in the database using **bcryptjs**.
   - Upon successful verification, a **JWT token** is generated using the user's unique ID and role.
3. **Token Generation:**
   - A JWT token is created using the **jsonwebtoken** library.
   - The token contains:
     - User ID (`_id`)
     - Role (`Renter` , `Owner` , or `Admin`)
   - The token is signed with a secret key stored securely in the `.env` file.
   - The token has an expiration time (e.g., 1 hour) to enhance security.
4. **Token Storage:**
   - The generated token is sent back to the client and stored in:
     - **LocalStorage** or **Cookies** on the frontend for session persistence.
   - The token is included in the `Authorization` header of subsequent API requests.

Authorization Mechanism

Authorization determines the level of access and actions a user is allowed to perform based on their role.

Middleware for Role-Based Access:

- **Auth Middleware:**
  - A middleware function (`authMiddleware.js`) verifies the JWT token sent with the API request.
  - If the token is valid, the user is granted access; otherwise, a `401 Unauthorized` response is returned.
- **Role-Based Access Control (RBAC):**
  - After verifying the token, the middleware checks the user's role to determine access to specific routes:
    - **Renter:**
      - Can view property listings, send booking requests, and check booking statuses.
    - **Owner:**
      - Can add, update, or delete properties and manage their availability status.
    - **Admin:**
      - Approves new owners, monitors platform activities, and enforces policies.

Technical Implementation Details

1. **Token Creation:**

```
const jwt = require('jsonwebtoken');

const generateToken = (id, role) => {
    return jwt.sign({ id, role }, process.env.JWT_SECRET, { expiresIn: '1h' });
};
```

2. **Authentication Middleware:**

```
const jwt = require('jsonwebtoken');
const User = require('../models/userModel');

const protect = async (req, res, next) => {
    let token;
    if (req.headers.authorization &&
req.headers.authorization.startsWith('Bearer')) {
        try {
            // Extract token
            token = req.headers.authorization.split(' ')[1];
            // Verify token
            const decoded = jwt.verify(token, process.env.JWT_SECRET);
            // Attach user info to request
            req.user = await User.findById(decoded.id).select('-password');
            next();
        } catch (error) {
            res.status(401).json({ message: 'Not authorized, token failed' });
        }
```

```
    } else {
        res.status(401).json({ message: 'No token, authorization denied' });
    }
};
```

3. **Role-Based Authorization:**

```
const authorize = ( ... roles) => {
    return (req, res, next) => {
        if (!roles.includes(req.user.role)) {
            return res.status(403).json({ message: 'Access forbidden: insufficient permissions' });
        }
        next();
    };
};
```

Security Measures

1. **Password Hashing:**
   - All passwords are hashed using **bcryptjs** before being stored in MongoDB.

```
const bcrypt = require('bcryptjs');
user.password = await bcrypt.hash(user.password, 10);
```

2. **Token Expiry:**
   - JWT tokens are configured to expire after a certain time (e.g., 1 hour), ensuring users reauthenticate periodically.
3. **Environment Variables:**
   - Sensitive data such as the JWT secret key is stored in a `.env` file and not hardcoded into the application.
4. **HTTPS:**
   - The app is designed to run on HTTPS in production to prevent token theft via insecure connections.
5. **CORS:**
   - The backend is configured with **CORS** to allow only specific domains to interact with the API.

Example Use Cases

1. **Renter Accessing Properties:**
   - API request: `GET /api/properties`
   - Authorization header: `Bearer <JWT_TOKEN>`

- Middleware verifies the token and grants access.

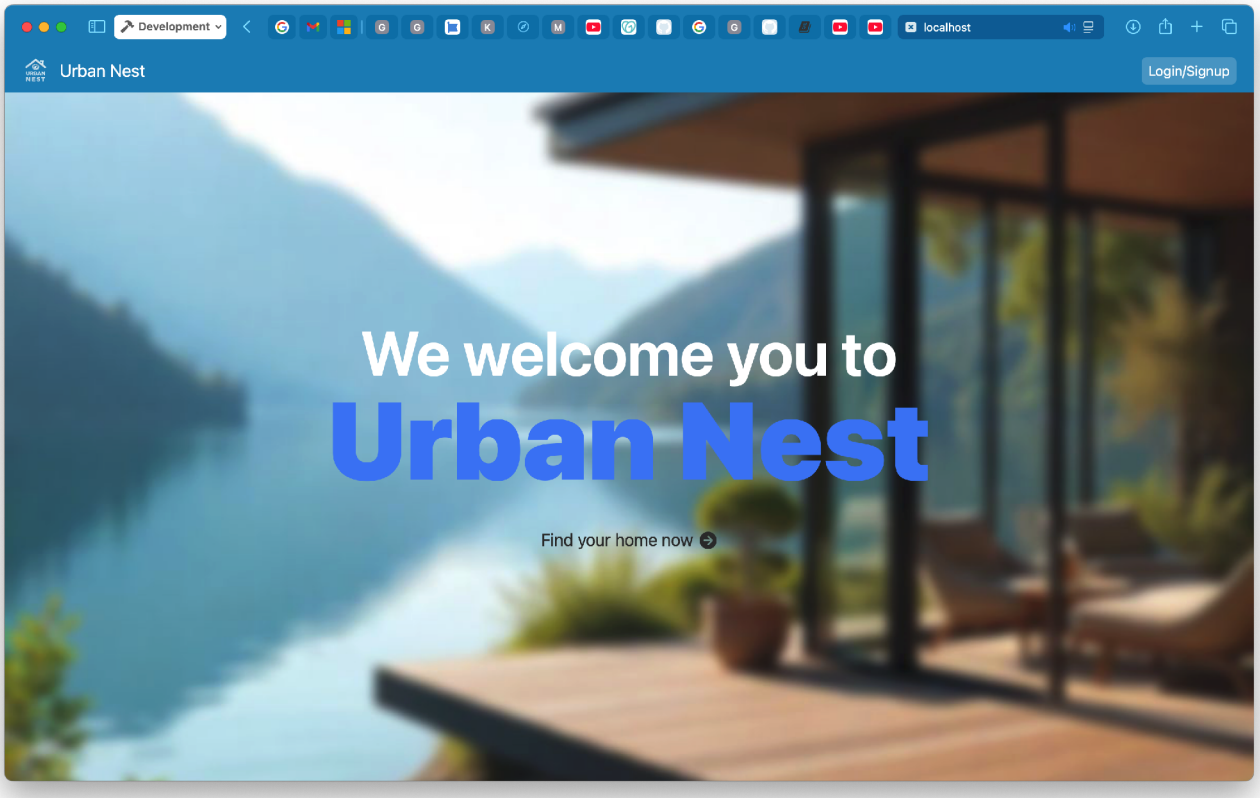
2. **Admin Approving an Owner:**
   - API request: `POST /api/admin/approve-owner`
   - Authorization header: `Bearer <ADMIN_JWT_TOKEN>`
   - Middleware checks for admin role and processes the request.

Conclusion

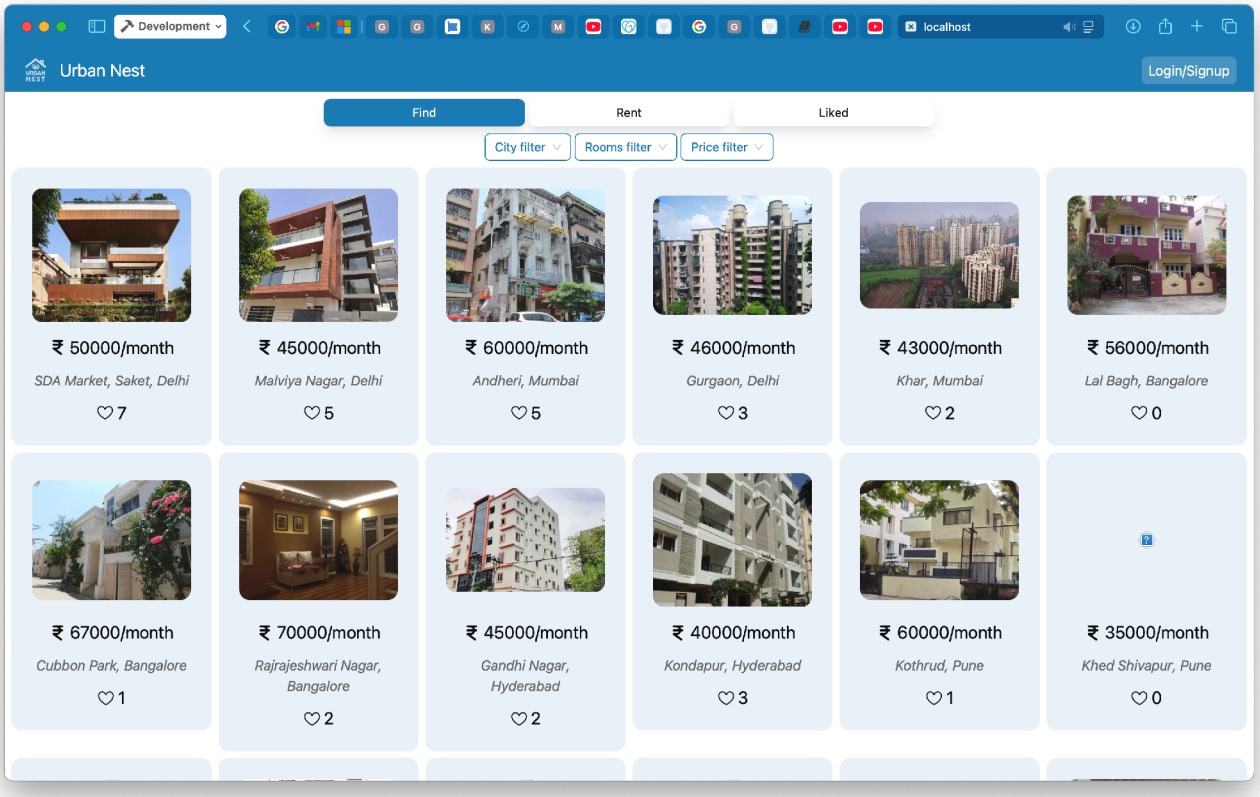
The Urban Nest implements a secure and efficient authentication and authorization system using **JWT tokens** for stateless session management and **middleware** for role-based access. This architecture ensures that user data remains secure and each role has appropriate permissions, contributing to a robust and scalable application.

User Interface

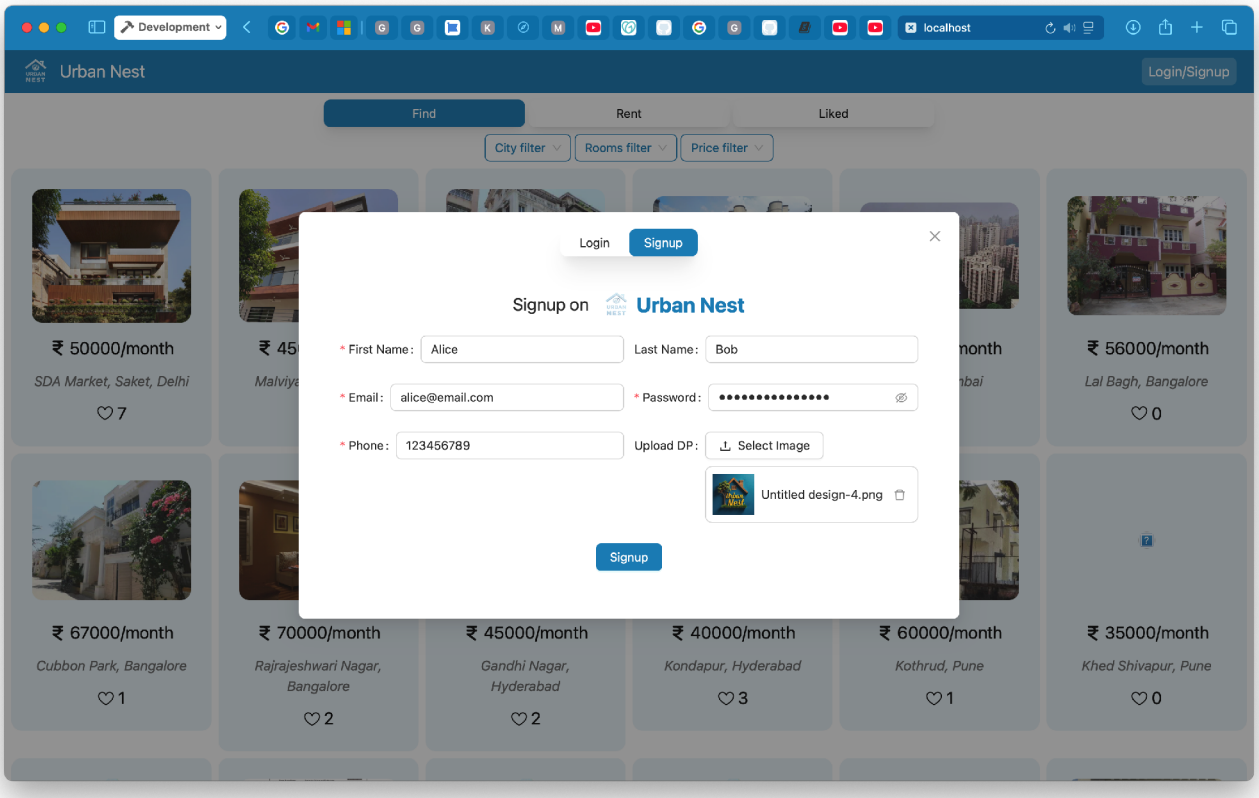
Welcome Screen



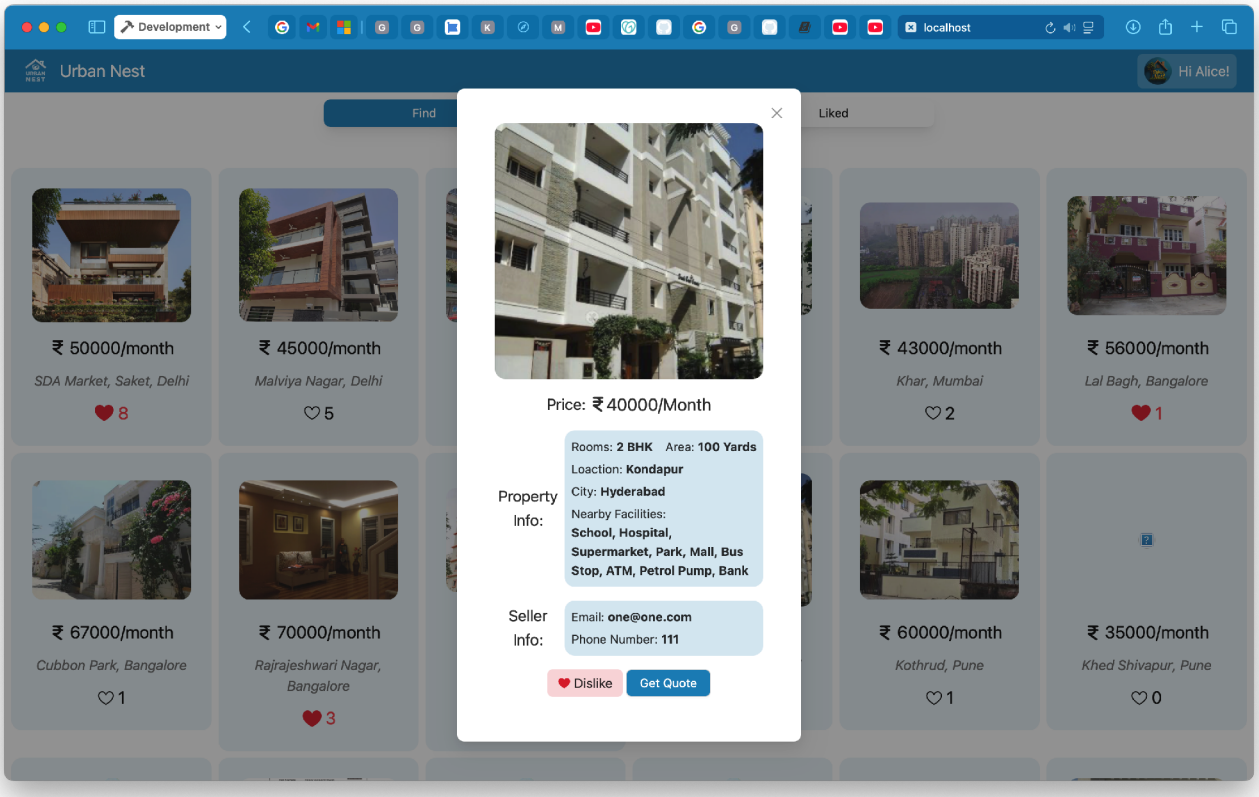
Dashboard



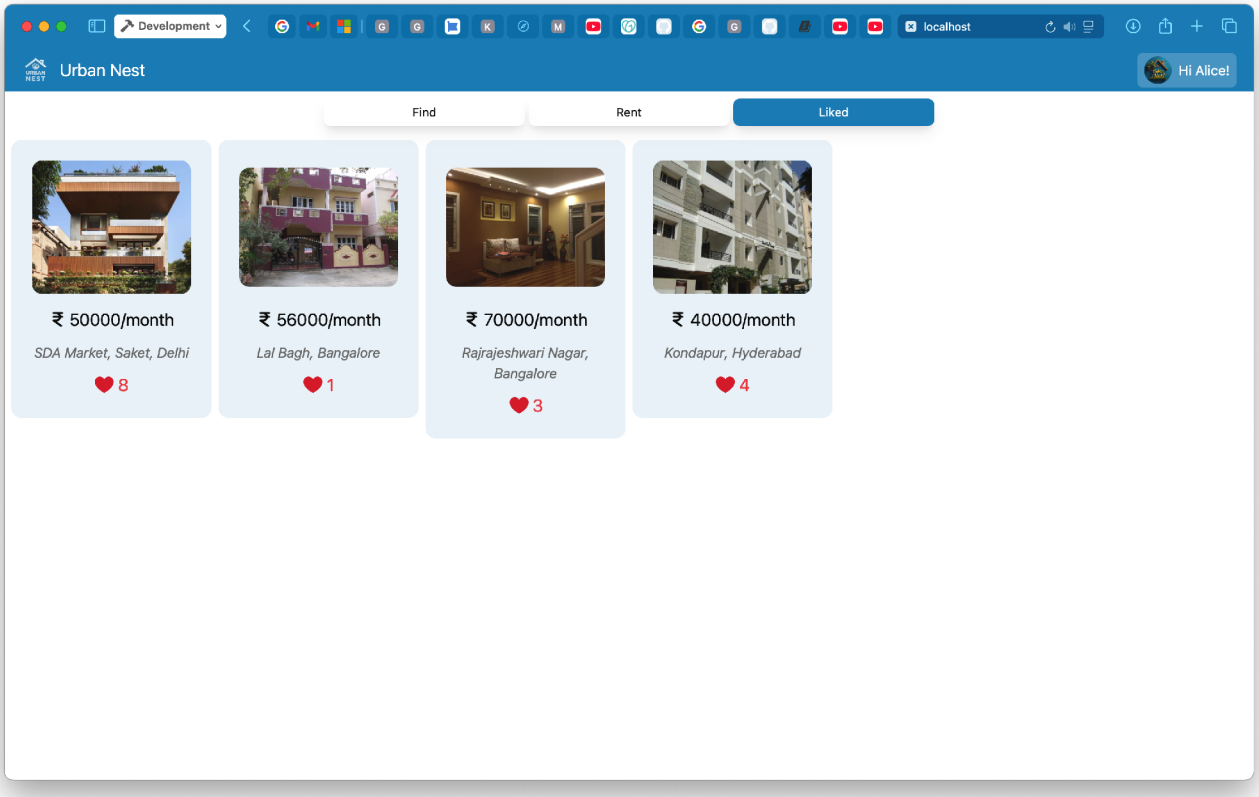
Login/Signup



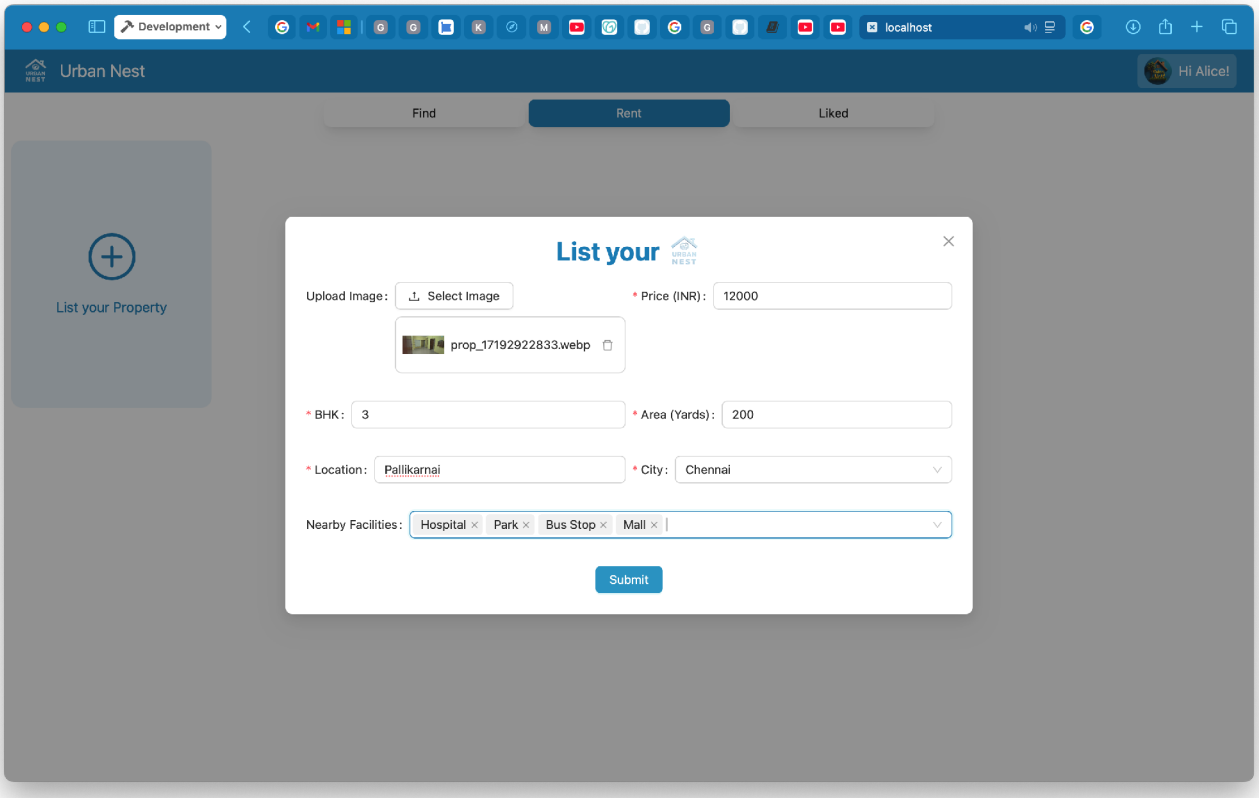
Liking Rental

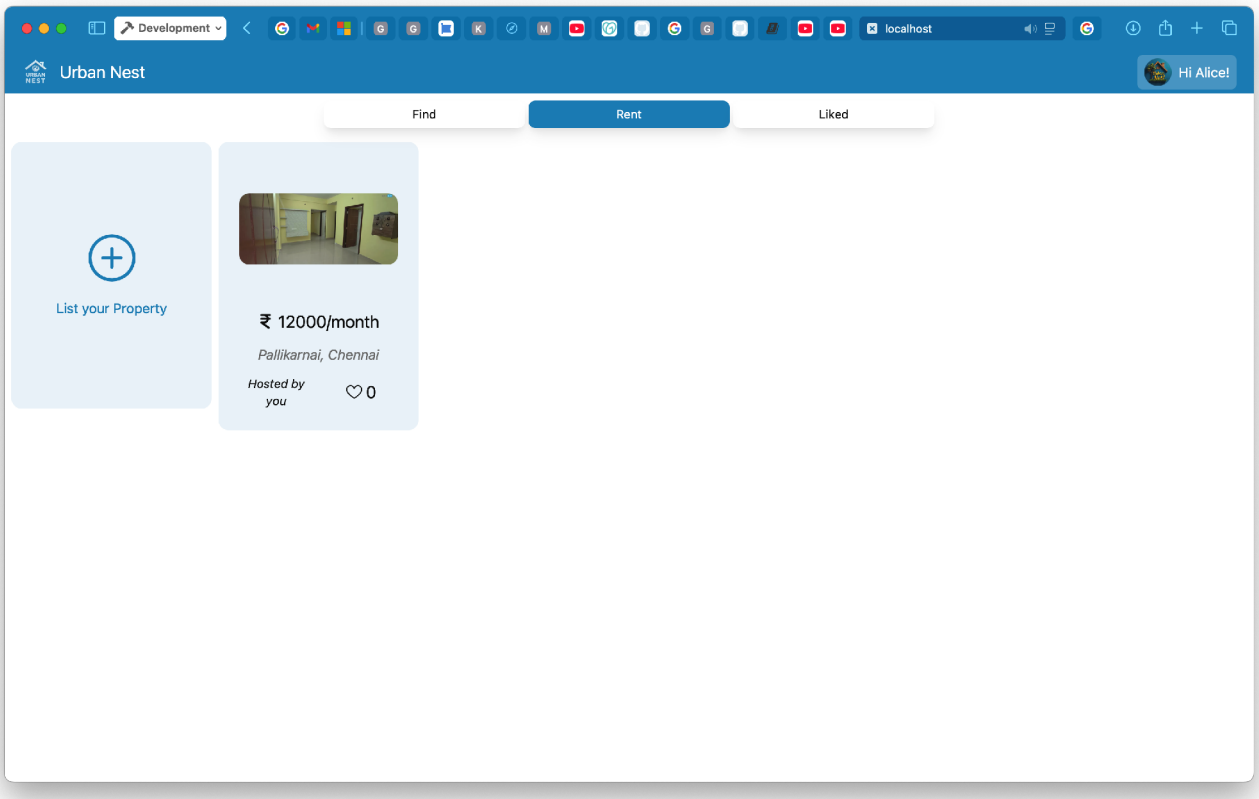


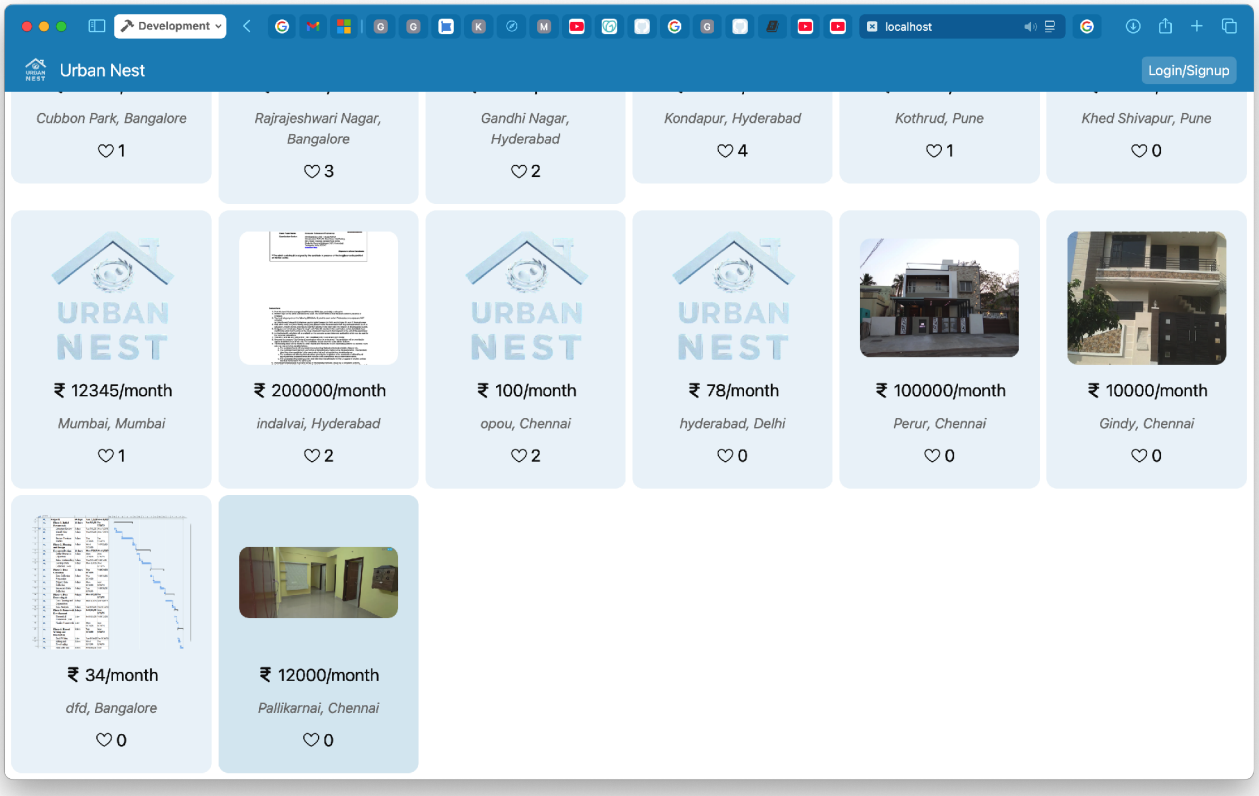
Liked Content



Listing your Home




Development
localhost
Urban Nest
Hi Alice!
Find
Rent
Liked
List your Property
₹ 12000/month
Pallikarnai, Chennai
Hosted by you
♡ 0


Development
localhost
Urban Nest
Login/Signup
Cubbon Park, Bangalore
♡ 1
Rajrajeshwari Nagar, Bangalore
♡ 3
Gandhi Nagar, Hyderabad
♡ 2
Kondapur, Hyderabad
♡ 4
Kothrud, Pune
♡ 1
Khed Shivapur, Pune
♡ 0
₹ 12345/month
Mumbai, Mumbai
♡ 1
₹ 200000/month
indalvai, Hyderabad
♡ 2
₹ 100/month
opou, Chennai
♡ 2
₹ 78/month
hyderabad, Delhi
♡ 0
₹ 100000/month
Perur, Chennai
♡ 0
₹ 10000/month
Gindy, Chennai
♡ 0
₹ 34/month
dfd, Bangalore
♡ 0
₹ 12000/month
Pallikarnai, Chennai
♡ 0

Known Issues

Here are some known issues or potential challenges that users or developers might encounter in the current version of the Urban Nest:

1. **Limited Search Optimization**:
   - Filters may not always return the most relevant results due to basic search algorithms.
   - Enhancing search with geolocation or advanced algorithms could improve the user experience.
2. **Incomplete Validation**:
   - User input validation (e.g., email format, password strength) might be inconsistent or insufficient across forms.
3. **Concurrency Issues**:
   - No mechanism to handle multiple renters booking the same property simultaneously. This could result in conflicts or double-booking.
4. **Image Upload Limitations**:
   - Image upload for properties might have limitations on file size or format, which could lead to failed uploads.
5. **Dependency on Admin Approval**:
   - All owner actions require admin approval, which might cause delays in property listings if the admin is unavailable.
6. **Security Concerns**:
   - Basic JWT-based authentication might be susceptible to token theft if not securely implemented.
   - Lack of rate limiting might leave the app vulnerable to brute-force attacks.
7. **Mobile Responsiveness**:
   - Some UI components might not render optimally on mobile devices, causing usability issues.
8. **Error Handling**:
   - Limited error messages for users. For instance, a failed booking attempt might not provide detailed reasons.
9. **Scalability**:
   - The app might face performance issues with a high number of concurrent users or properties due to the absence of load balancing or database optimization.

Future Enhancements

Potential features and improvements to consider in future versions:

1. **Enhanced Search and Recommendations**:

- Implement AI/ML algorithms to suggest properties based on user preferences and browsing history.

2. **Push Notifications**:
   - Add notifications for booking updates, admin approvals, or changes in property status.
3. **Advanced Admin Dashboard**:
   - Develop a more sophisticated admin interface with analytics and tools for better platform management.
4. **Integrated Payment System**:
   - Include secure payment gateways to handle rental payments and deposits within the app.
5. **Multi-language Support**:
   - Add language options to cater to a diverse user base.
6. **Tenant and Owner Ratings**:
   - Introduce a rating and review system for both renters and owners to enhance transparency and trust.
7. **Chat System**:
   - Implement real-time chat between renters and owners for faster communication.
8. **Geolocation Integration**:
   - Allow users to view properties on a map and filter results based on proximity to certain locations.
9. **Improved Security Measures**:
   - Use HTTPS for all connections, enforce secure password policies, and implement two-factor authentication (2FA).
10. **Property Scheduling System**:
    - Enable renters to schedule property visits directly through the app.
11. **Mobile Application Development**:
    - Develop native apps for iOS and Android to improve accessibility and user experience.
12. **Offline Mode**:
    - Allow basic functionality, such as browsing previously loaded properties, without an active internet connection.

These enhancements and fixes will address current limitations and add value, ensuring a more robust and user-friendly experience for all users.