Table Of Content

1. Project Title:

" HorizonHomes: Smart Property Discovery & Management Platform "

2. Project Overview:

Horizon Homes is an innovative, full-stack MERN (MongoDB, Express, React, Node.js) web application designed to simplify and streamline the process of searching, listing, and reserving residential properties. The primary goal was to create a comprehensive, end-to-end solution that empowers users at every stage of the real estate transaction process, from browsing properties to finalizing bookings and making payments. This platform is intended to address the evolving needs of the real estate market, providing a seamless and user-friendly experience for both potential buyers/renters and property owners.

The application enables secure user authentication and role-based access control, distinguishing between visitors, agents, and admins. Visitors can browse available properties, utilize a mortgage calculator, and explore neighborhood insights, while agents can list new properties, manage their own listings, and facilitate interactions with potential clients. Admins have the highest level of access, overseeing all property listings, approving or rejecting bookings, managing user roles, and processing payments. By using JWT (JSON Web Tokens) for secure login and role management, the platform ensures a robust and safe experience for all users.

Horizon Homes facilitates real-time operations on property data, enabling agents and admins to create, read, update, and delete (CRUD) property listings with immediate reflections across the platform. Additionally, the app integrates an online booking system where users can reserve properties and make payments securely. The booking process supports payment intents, ensuring that all transactions are processed safely and efficiently.

Beyond the basic real estate functionalities, the application also features a mortgage calculator, which helps potential buyers assess their financial capabilities by providing estimates on monthly payments based on property prices, down payments, interest rates, and loan terms. It also includes neighborhood insights, offering users valuable information about the local area, such as proximity to schools, public transportation, amenities, and safety ratings.

Designed with a mobile-first approach, the platform is fully responsive and optimized for seamless performance across a wide range of devices, from smartphones and tablets to desktop computers. The front-end of the application is built using React, which powers dynamic user interfaces, while the back-end is developed using Node.js and Express, ensuring fast, efficient handling of requests and smooth communication with the database, MongoDB.

The project also emphasizes security with encrypted password storage and role-based access controls that protect sensitive information. Additionally, the application integrates real-time data management, ensuring that updates to listings, bookings, and user data are instantly reflected in the UI.

With features like secure login, dynamic property management, online booking, payment processing, and useful utilities like mortgage calculators and neighborhood insights, Horizon Homes is an all-in-one platform that provides an efficient and modern solution for the real estate industry, making it easier for users to find, book, and manage properties in a secure and intuitive manner.

3. Technologies Used:

1. Frontend Technologies:

- HTML5: The foundational markup language for creating the structure of web pages. HTML5 provides the basic skeleton for the entire user interface, supporting modern web standards like semantic elements (`<header>`, `<footer>`, `<article>`, etc.) and multimedia content.
- CSS3: The styling language used to define the look and feel of the web pages. CSS3 allows for responsive design (using media queries) and styling of elements like layout, typography, and animations. It's used to create a clean, modern, and visually appealing user interface.
- React: A JavaScript library used for building the front-end of the application. React allows the creation of dynamic and interactive user interfaces with components. It uses a virtual DOM to efficiently update the UI based on user interactions, making it fast and responsive. The component-based architecture simplifies maintenance and scaling of the application.
- Vite: A modern, fast build tool used to bundle and serve the application in development. Vite optimizes the development experience by providing fast hot module replacement (HMR) and quick build times. It is a powerful alternative to older build tools like Webpack, making the development process smoother and more efficient.
- Bootstrap (or Custom CSS): A front-end framework or custom CSS can be used for creating responsive layouts and user interfaces. Bootstrap provides pre-built components (such as navigation bars, modals, and buttons) that help speed up UI development. Custom CSS allows for more flexibility and uniqueness in styling the app.
- Axios: A promise-based HTTP client used for making API calls to the backend. Axios is used to communicate with the backend to fetch property data, submit booking requests, and handle user authentication. It's simple to integrate into React components and helps manage API interactions.

2. Backend Technologies:

- Node.js: A runtime environment that allows JavaScript to be executed on the server-side. Node.js is built on the V8 JavaScript engine, making it fast and efficient. It’s non-blocking, asynchronous, and event-driven, which makes it ideal for handling concurrent requests in a real-time application like Horizon Homes.
- Express: A minimal and flexible web application framework for Node.js. Express simplifies routing, middleware management, and handling HTTP requests and responses. It enables the creation of RESTful APIs for interactions between the frontend and backend, such as fetching property listings, booking data, and processing user authentication.

- MongoDB: A NoSQL database used to store data in a flexible, JSON-like format called BSON (Binary JSON). MongoDB is well-suited for this project because it can handle dynamic, unstructured data, which is often the case with property listings (e.g., different fields for different types of properties). It allows for fast querying, scalability, and ease of data manipulation.
- Mongoose: An Object Data Modeling (ODM) library for MongoDB that provides a schema-based solution to interact with the database. Mongoose allows defining models with specific data types, validation, and relationships between entities (e.g., users, properties, bookings). It abstracts much of the boilerplate code involved in querying MongoDB, making the interaction smoother and more manageable.
- JWT (JSON Web Token): A standard for securely transmitting information between parties as a JSON object. In Horizon Homes, JWT is used for user authentication and authorization. After a user logs in, the backend generates a token that is sent to the client. This token is then used for authenticating API requests and ensuring that the user has the proper permissions based on their role (Visitor, Agent, Admin).

3. Security Features:

- JWT Authentication: Used for secure user login and maintaining session information. After a user logs in, they receive a token that can be used to authenticate subsequent requests. This ensures that sensitive data is only accessible to authorized users (e.g., Admins, Agents).
- CORS (Cross-Origin Resource Sharing): Middleware that allows the backend to specify which domains are allowed to make requests to it. This is important when the frontend and backend are hosted on different domains or ports, as it ensures that only authorized sources can access the backend APIs.

4. Development and Build Tools:

- Vite: As mentioned, Vite is used as a modern build tool for the frontend. It provides fast and efficient development with hot module replacement, quick build times, and optimized production builds. These speeds up the development process and allows developers to see changes in real-time without having to refresh the browser.
- Git and GitHub: Git is used for version control, while GitHub hosts the repository for collaboration and code management. Git enables the tracking of changes, handling of merge conflicts, and collaboration among multiple developers working on the project.

5. Deployment and Hosting:

- MongoDB Atlas: A cloud-based MongoDB service that can be used to host the MongoDB database. MongoDB Atlas provides automatic backups, scalability, and monitoring, making it a great choice for production environments.

6. Utilities and Additional Features:

- Mortgage Calculator: A front-end utility that allows users to estimate monthly mortgage payments based on the price of a property, down payment, interest rate, and loan term.
- Neighborhood Insights API: An external API or dataset integrated into the application to provide users with information about the surrounding area of a property (e.g., schools, public transport, amenities, etc.).

These technologies work together to create a robust, scalable, and secure platform that delivers an intuitive and user-friendly experience for both front-end users and back-end administrators. The integration of various tools and libraries ensures that Horizon Homes is not only functional but also secure, efficient, and maintainable.

4. Project Folder Structure of Horizon Homes:

The Horizon Homes project follows a structured and organized folder architecture to ensure scalability, maintainability, and separation of concerns. The folder structure is divided primarily into two main sections: the backend and client directories, each handling distinct responsibilities for the project.

Here's a detailed explanation of the folder structure:

---

1. Root Folder Structure:

```
/real-estate
  ├── backend/
  ├── client/
  ├── package.json
  ├── setup.sh
  ├── start.sh
  └── README.md
```

- `/real-estate/`: The root directory contains the entire project and houses both the backend and frontend code. It also includes configuration files for both parts of the application.
    - `package.json`: A central configuration file for the entire project. It contains metadata, scripts, and dependencies for both the backend and frontend. The dependencies here could include Express, Node.js, and other shared dependencies for the backend, while also listing frontend dependencies if they are part of the build process.
    - `setup.sh` and `start.sh`: Shell scripts that are typically used for setting up and starting the application, respectively. The `setup.sh` script may install the necessary dependencies and configure the environment, while `start.sh` is responsible for launching the backend or the entire application.
    - `README.md`: A markdown file that contains documentation about the project, explaining its purpose, setup instructions, and how to run the application.

---

2. Backend Folder Structure:

```
/backend
  ├── .env
  ├── middleware/
  ├── models/
  ├── node_modules/
  ├── package-lock.json
  ├── package.json
  ├── routes/
  ├── seedData.js
  └── server.js
```

- `/backend/`: This folder contains all the server-side code, responsible for handling the business logic, database interactions, and API endpoints.
    - `.env`: This file holds environment variables, such as database connection strings, API keys, or other sensitive configuration details. It is essential for securing sensitive data, especially for production environments. The `dotenv` library typically loads these variables when the server starts.
    - `middleware/`: This directory contains middleware functions used by the Express server. Middleware functions can be used for various tasks such as request logging, authentication, error handling, CORS configuration, etc. It might include files for validating user inputs, handling errors, or authorizing users based on their roles.
    - `models/`: This folder defines the Mongoose models that represent the entities in the database (e.g., User, Property, Booking). Each model is associated with a specific MongoDB collection and defines the schema for how data should be structured and validated. For example, `User.js`, `Property.js`, and `Booking.js` models may be found here, outlining the fields, data types, and relationships between entities.
    - `node_modules/`: This folder contains all the Node.js dependencies installed via `npm` (Node Package Manager). These dependencies include libraries such as Express, Mongoose, jsonwebtoken (JWT), and bcrypt.
    - `package-lock.json`: This file locks the versions of dependencies to ensure consistent installations across different environments. It is automatically generated when `npm install` is executed.
    - `package.json`: The `package.json` file in the backend contains metadata about the backend server and lists backend-specific dependencies (such as Express, Mongoose, bcrypt, etc.). It also contains scripts for running the server or building the backend API.
    - `routes/`: This folder defines the API endpoints that the backend exposes. Each file in this directory may correspond to a specific set of routes related to an entity. For instance:
        - `auth.js`: Contains routes for user authentication, such as login and registration.
        - `properties.js`: Contains routes for managing property listings (CRUD operations).
        - `bookings.js`: Contains routes for managing bookings, such as creating, updating, and deleting bookings.

- `seedData.js`: This file is typically used to seed the database with sample or initial data. This is useful for testing or when launching the application for the first time to populate the database with sample properties, users, or bookings.
- `server.js`: This is the main entry point for the backend application. It sets up the Express server, configures middleware (e.g., body parsers, CORS), defines routes, connects to the database, and listens for incoming requests. It integrates all the backend functionalities and is where the server is started.

---

3. Client Folder Structure:

```
/client
   ├── .gitignore
   ├── eslint.config.js
   ├── images/
   ├── index.html
   ├── node_modules/
   ├── package-lock.json
   ├── package.json
   ├── public/
   ├── README.md
   ├── src/
   └── vite.config.js
```

- `/client/`: This folder contains all the client-side code, which is responsible for rendering the user interface and interacting with the backend APIs.
  - `.gitignore`: A file used to specify which files and directories Git should ignore. This is useful for excluding sensitive data, configuration files, or dependencies that do not need to be tracked by version control.
  - `eslint.config.js`: This configuration file is used by ESLint, a tool for identifying and fixing problems in JavaScript code. It enforces coding standards and ensures code quality.
  - `images/`: This directory contains static image files used in the application, such as property photos, logo images, and other visual assets.
  - `index.html`: The main HTML file that serves as the entry point for the client-side application. This file contains the basic structure of the page and includes links to JavaScript files that are needed to run the React app.
  - `node_modules/`: This folder contains all the dependencies for the client-side React application. These dependencies include libraries like React, ReactDOM, React Router, and Axios.
  - `package-lock.json`: Similar to the backend, this file locks the versions of dependencies used by the client to ensure consistent installations across different environments.
  - `package.json`: This file contains metadata about the client-side application, including dependencies, scripts, and configuration details. It lists the required libraries for React and other front-end technologies.

- `public/`: This folder contains static files that are publicly accessible and can be served directly by the server. It may include files such as the favicon, images, or any other public assets.
- `README.md`: The markdown file that contains information about the client-side application, including setup instructions, development guidelines, and an overview of the project.
- `src/`: The `src/` folder holds the source code for the front-end application. This folder is the heart of the React app and contains various components, utilities, and styles. Typical subfolders in `src/` might include:
  - `components/`: Contains React components like headers, footers, property cards, and booking forms.
  - `contexts/`: This folder could include React Context providers, which manage the global state for the application, such as user authentication, property listings, and bookings.
  - `services/`: Contains services or API functions for interacting with the backend, such as fetching property data, posting a booking, or submitting user credentials.
  - `utils/`: Utility functions, such as those for formatting data, calculating mortgage payments, or validating input fields.
  - `App.jsx`: The main component that serves as the root of the React application. It renders the primary UI components and manages the overall application flow.
  - `index.jsx`: The entry point for the React application that renders the root `App` component into the DOM.
- `vite.config.js`: This configuration file is used by Vite to optimize and bundle the front-end application. It specifies build settings, plugin configurations, and any special behavior needed for production builds.

The Horizon Homes project has a well-organized folder structure that adheres to the principles of separation of concerns. The backend and frontend are clearly separated into distinct directories (`backend/` and `client/`), making the project easy to maintain and scale.

- The backend directory handles server-side logic, API routes, and database interactions.
- The client directory is responsible for rendering the user interface, making API calls, and managing client-side functionality using React.
- Shared configuration files, like `package.json` and environment variables, ensure smooth integration between the two parts of the application.

5. Main Features:

The Horizon Homes web application is a comprehensive real estate platform designed to simplify the process of searching, listing, and reserving residential properties. It incorporates a wide array of features that aim to enhance the user experience, streamline property management, and ensure

seamless interactions between buyers, agents, and admins. The main features of the project are outlined below:

1. Property Listing: Real estate agents or property owners list their properties on the website. They provide details such as property type (apartment, house, commercial space), location, size, amenities, price, and photos.

2. Search and Filters: Users (potential buyers or renters) browse the website and use search filters (like location, etc.) to find properties that match their preferences.

3. Property Details: Users can view detailed information about the properties, including descriptions, photos, floor plans, amenities, nearby facilities (schools, hospitals, etc.), and contact details of the agent or owner.

4. Booking or Inquiry: Interested users can inquire about the property or, in some cases, directly book a viewing or make an appointment to visit the property. They may also have the option to schedule virtual tours.

5. Communication: The website facilitates communication between buyers/renters and sellers/agents. It may include messaging systems, contact forms, or direct contact information exchange to arrange property viewings, negotiate terms, or ask questions.

6. Booking Process: If the buyer decides to proceed, they might have the option to book the property online, paying a reservation fee or deposit through secure payment gateways. Contracts or agreements might be initiated or signed digitally.

- Book Property Visit process

1. User Login Process: Users access the website by creating an account or logging in. They provide their credentials (username/email and password) to gain access to personalized features like saving favorite properties, , or managing their profile information.

2. Property Searching Process: Users can search for properties using various filters such as location, etc. The website displays a list of properties matching the specified criteria, allowing users to view details, photos, and contact information for each property.

3. Registering New User Process: New users can register by providing necessary details like name, email address, and creating a password. Some websites might require additional information for verification purposes. After registration, users gain access to the website's functionalities.

4. Add Property Process: Property owners or real estate agents can add their properties to the website by providing detailed information about the property. This includes property type, location, size, photos, amenities, price, and any other relevant information. They might also upload documents like property deeds or floor plans.

5. Book Property Visit Process: Users interested in a particular property can schedule a visit or viewing through the website. They may have the option to select a convenient date for the visit. The website might facilitate communication between the user and the property owner/agent to confirm the visit and provide any additional details.

- Secure User Authentication and Role-Based Access Control

One of the core features of Horizon Homes is its secure user authentication system. The platform supports multiple types of users with different roles and access levels:

- Visitor: A non-logged-in user who can browse available properties, view basic property information, use the mortgage calculator, and access neighborhood insights.
- Agent: A registered user who can list new properties, edit or update existing property listings, view booking requests, and manage interactions with potential buyers or renters.
- Admin: The highest level of user access, with full administrative rights. Admins can manage users (including agents), approve or reject property listings, oversee bookings, manage transactions, and handle system-wide configurations.

The platform uses JWT (JSON Web Tokens) for authentication, ensuring that only authorized users can access their specific roles.

- Real-Time Property Listings and Management

The platform enables agents and admins to manage property listings with ease. Key features include:

- Real-Time Property Listings: Agents and admins can create, read, update, and delete (CRUD) property listings in real-time. This allows for immediate updates to property data, ensuring that potential buyers or renters always see the latest available properties.
- Property Details: Each property listing includes detailed information such as the title, description, price, location, images, and availability status. Listings can also include additional features such as amenities, floor plans, and neighborhood information.
- Admin Approval: Admins can approve or reject property listings submitted by agents to ensure quality control and prevent fraudulent or inappropriate listings.

The property data is stored and managed in MongoDB, making it easy to store large amounts of unstructured data with flexible schema definitions.

- Advanced Search and Filtering Capabilities

The search and filtering system in Horizon Homes is designed to help users easily find properties that meet their specific needs. The platform provides:

- Property Search: Users can search for properties based on keywords, such as location, price range, type of property (e.g., apartment, house, studio), or amenities (e.g., swimming pool, garage).
- Advanced Filters: Users can refine their search using various filters, such as price, size, availability, and other criteria. This ensures that users can quickly find the most relevant properties without having to scroll through irrelevant listings.
- Location-Based Search: The search tool allows users to search by city or neighborhood, ensuring that location-based criteria are prioritized.

These features ensure that visitors and agents can find the right properties quickly, improving the overall user experience.

- Mortgage Calculator

A key utility for users is the Mortgage Calculator, which helps potential buyers estimate their monthly mortgage payments. This tool allows users to:

- Input Parameters: Users can input the property price, down payment amount, interest rate, and loan term (e.g., 15 or 30 years).
- Calculate Monthly Payment: The mortgage calculator computes the estimated monthly payments based on the provided information, helping users better understand their financial obligations before making a purchase.

This tool is especially useful for buyers to assess whether a property fits within their budget, providing transparency and facilitating decision-making.

- Neighborhood Insights

In addition to detailed property data, Horizon Homes offers valuable neighborhood insights for each listing:

- Local Amenities: Users can view nearby amenities such as schools, grocery stores, parks, and hospitals, helping them evaluate the convenience and lifestyle of the neighborhood.
- Transportation Information: Information about public transportation options, such as bus stops and metro stations, is provided to help users assess the ease of commuting.
- Safety Ratings: Users can see neighborhood safety data, which includes crime rates and safety scores based on local crime statistics.

This information is especially valuable for users who want to ensure that the neighborhood aligns with their personal preferences and lifestyle.

- Admin Dashboard

The Admin Dashboard is a powerful tool that allows administrators to manage the platform and its users effectively. Key functionalities of the dashboard include:

- User Management: Admins can view and manage all user accounts, including visitors, agents, and other admins. They can modify roles, deactivate accounts, and track user activity.
- Property Management: Admins can oversee all property listings, including approving or rejecting new listings, editing existing listings, and removing outdated or inappropriate listings.
- Booking and Payment Management: Admins have full visibility into the booking and payment system. They can approve bookings, track payment statuses, and monitor transactions.
- Analytics and Reports: The dashboard provides insights into user activity, property popularity, booking trends, and other performance metrics, helping admins make data-driven decisions.

This comprehensive dashboard gives admins full control over the platform, making it easy to maintain and monitor the system.

- Security Features

Security is a critical aspect of the Horizon Homes platform. Several measures are in place to protect sensitive user data and prevent unauthorized access:

- JWT Authentication: User login and authentication are handled securely using JWT tokens. Tokens are generated on login and used to authenticate requests, ensuring that only authorized users can access specific functionalities.
- Role-Based Access Control: Users are assigned specific roles (visitor, agent, admin) with different levels of access. This ensures that users can only access the functionality that is relevant to their role.

These security features help protect user data, ensure privacy, and provide peace of mind for both users and administrators.

6. Code Snippets with Explanations:

Java script is a scripting language used to enhance the functionality of the browser. Java script is integrated with HTML and CSS. Java script facilitates the developer with properties related to document windows, frames, loaded documents and links.

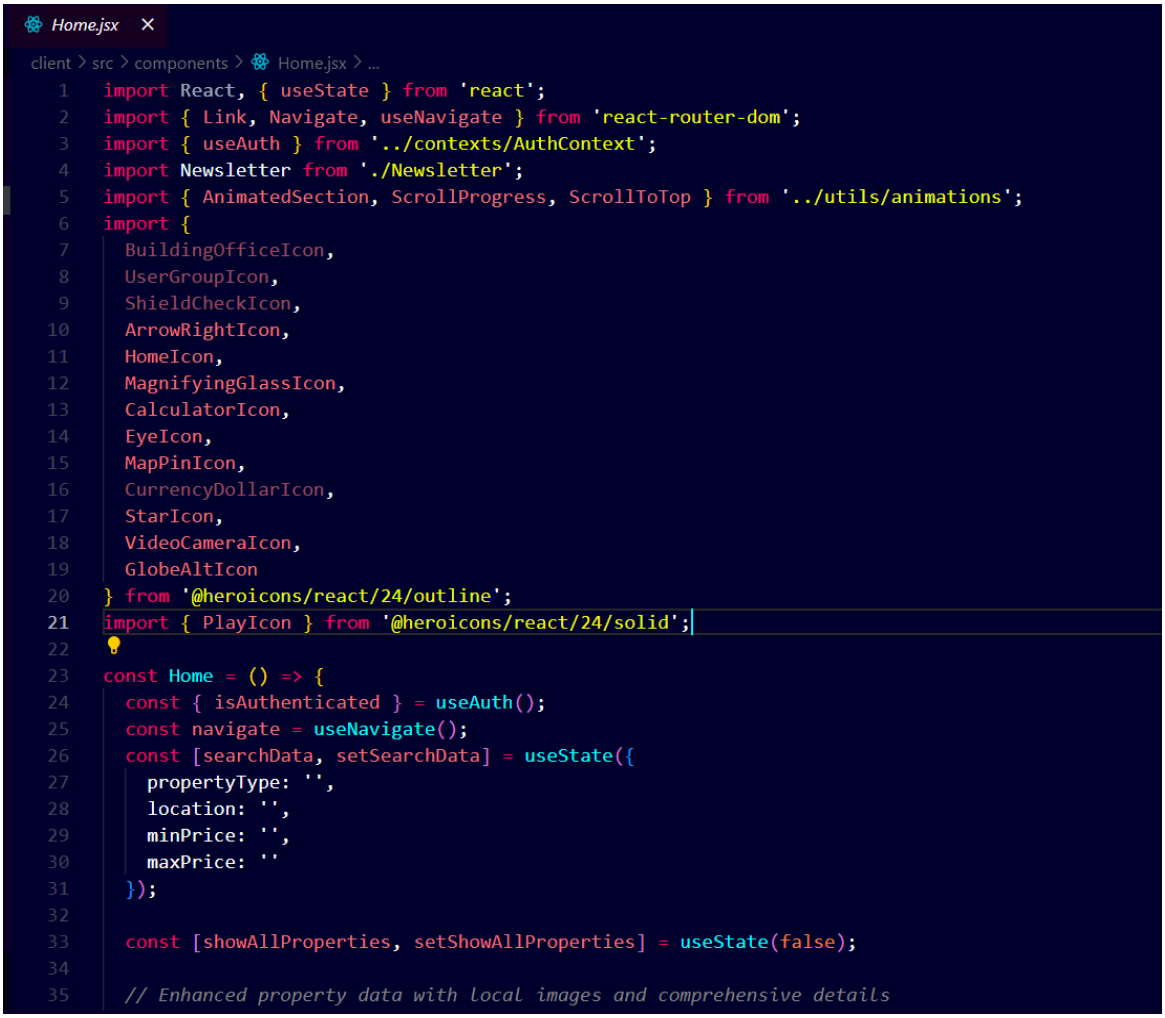
React is a popular JavaScript library used for building user interfaces. It is used for frontend due to its flexibility, reusability, and ease of use.

For the Back-end part we will use Node.js which can generate dynamic page content. It also helps to open , read, write and close files on the server and collect Login Authentication data. Node.js can add, delete, modify data in your database.

For Data Storage, we use MongoDB(Atlas, Compass), it is a popular NoSQL database it can handle large amounts of data and high traffic loads.

- Client End:

Home.jsx



```jsx
import React, { useState } from 'react';
import { Link, Navigate, useNavigate } from 'react-router-dom';
import { useAuth } from '../contexts/AuthContext';
import Newsletter from './Newsletter';
import { AnimatedSection, ScrollProgress, ScrollToTop } from '../utils/animations';
import {
  BuildingOfficeIcon,
  UserGroupIcon,
  ShieldCheckIcon,
  ArrowRightIcon,
  HomeIcon,
  MagnifyingGlassIcon,
  CalculatorIcon,
  EyeIcon,
  MapPinIcon,
  CurrencyDollarIcon,
  StarIcon,
  VideoCameraIcon,
  GlobeAltIcon
} from '@heroicons/react/24/outline';
import { PlayIcon } from '@heroicons/react/24/solid';

const Home = () => {
  const { isAuthenticated } = useAuth();
  const navigate = useNavigate();
  const [searchData, setSearchData] = useState({
    propertyType: '',
    location: '',
    minPrice: '',
    maxPrice: ''
  });

  const [showAllProperties, setShowAllProperties] = useState(false);

  // Enhanced property data with local images and comprehensive details
```

Explanation:

The code defines the Home component for the Horizon Homes real estate application. It includes a search form for filtering properties by type, location, and price. The homepage features a video background, a hero section with a call-to-action, and a featured properties section that displays property details like price, location, and amenities. Users can view more properties or take virtual tours. Additional sections include services offered by Horizon Homes (like property sales, rentals, and mortgage assistance), key stats such as properties sold and happy clients, and client testimonials with ratings. The footer provides links to other pages, contact info, and social media. The component also features a ScrollProgress bar and a ScrollToTop button for easy navigation. The design ensures a responsive, user-friendly experience with dynamic content, and uses React hooks for state management and navigation.

Login.jsx

Explanation:

The Login component is a React form used for authenticating users in the Horizon Homes application. It allows users to enter their email and password to sign in. The component uses useState to manage form data, error messages, and loading state. It also allows users to toggle password visibility with an eye icon. When the form is submitted, it calls the login function from the `useAuth` context to authenticate the user and navigate them to the dashboard if successful. The layout features a gradient background with animated elements and a clean, modern design. The login form is displayed in a backdrop-blurred container with fields for email and password. If authentication fails, an error message is shown. There's a link for users to navigate to the registration page if they don't have an account. Additionally, the login button is disabled during loading, and a spinner is displayed to indicate the ongoing sign-in process. The form includes security-related visual cues like SSL encryption and privacy protection indicators.

Register.jsx

```jsx
import React, { useState } from 'react';
import { Link, useNavigate } from 'react-router-dom';
import { useAuth } from '../contexts/AuthContext';
import { EyeIcon, EyeSlashIcon, HomeIcon, UserIcon, LockClosedIcon, EnvelopeIcon, UserCircleIcon } from '@

const Register = () => {
  const [formData, setFormData] = useState({
    name: '',
    email: '',
    password: '',
    confirmPassword: '',
    role: 'user',
  });
  const [error, setError] = useState('');
  const [loading, setLoading] = useState(false);
  const [showPassword, setShowPassword] = useState(false);
  const [showConfirmPassword, setShowConfirmPassword] = useState(false);

  const { register } = useAuth();
  const navigate = useNavigate();

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    setError('');

    if (formData.password !== formData.confirmPassword) {
      setError('Passwords do not match');
      return;
```

Explanation:

The Register component is a form that allows users to create a new account on the Horizon Homes platform. It includes fields for the user's name, email, password, confirm password, and account type (user or admin). The form validates the password strength and ensures that the passwords match before submitting the registration data. If any errors occur, they are displayed to the user. The component also includes a password visibility toggle for both the password and confirm password fields. A loading spinner is shown when the registration is in progress, and once successful, the user is navigated to the dashboard. Additionally, the page has animated background elements, a responsive design with icons, and a clean UI, showcasing key features of the platform, such as browsing properties, connecting with agents, and using a secure platform. It also includes links for users to log in if they already have an account and highlights security features like SSL encryption and privacy protection.

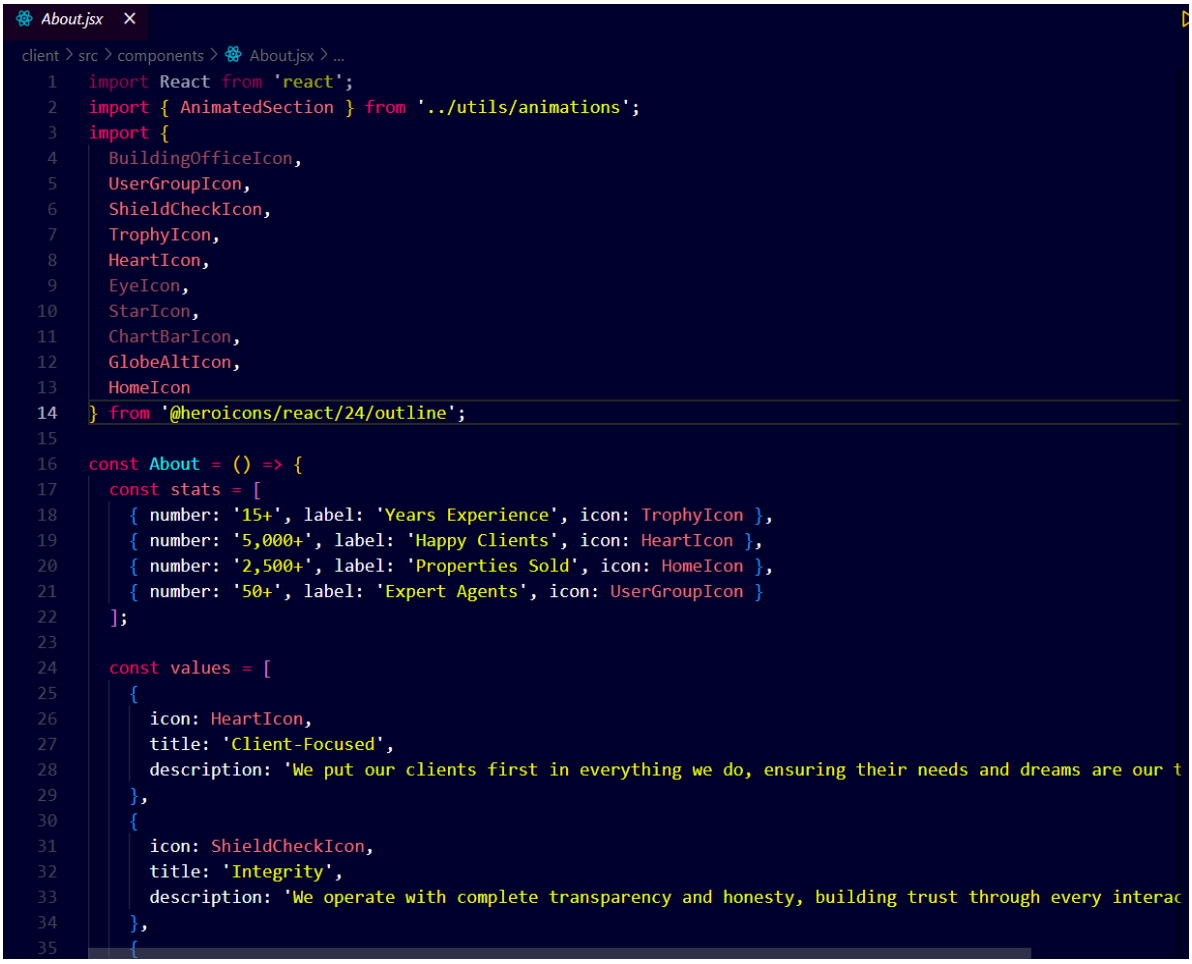
Admin.jsx

```jsx
import React, { useEffect, useState } from 'react';
import { bookingAPI, propertyAPI } from '../services/api';
import {
  UserIcon,
  CurrencyDollarIcon,
  CalendarIcon,
  CheckCircleIcon,
  ClockIcon,
  HomeIcon,
  EyeIcon,
  HeartIcon,
  ChartBarIcon,
  UsersIcon,
  BuildingOfficeIcon,
  DocumentTextIcon,
  PlusCircleIcon,
  XMarkIcon,
  PencilIcon,
  TrashIcon,
  MagnifyingGlassIcon,
  FunnelIcon,
  ArrowDownIcon,
  ArrowUpIcon,
  ExclamationTriangleIcon
} from '@heroicons/react/24/outline';

const AdminDashboard = () => {
  const [activeTab, setActiveTab] = useState('overview');
  const [bookings, setBookings] = useState([]);
  const [properties, setProperties] = useState([]);
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState('');
  const [searchTerm, setSearchTerm] = useState('');
  const [filterStatus, setFilterStatus] = useState('');
```

Explanation:

The AdminDashboard component is the main dashboard for administrators managing a real estate platform. It provides tabs for managing Overview, Bookings, Properties, and Users, each displaying relevant data and allowing various administrative actions. The dashboard fetches and displays sample data for bookings, properties, and users. It allows admins to filter, search, and manage these entities, including updating booking statuses and deleting records. It also includes a modal for confirming actions like deleting items. The Overview tab shows key statistics such as total bookings, active properties, and total revenue, while other tabs allow detailed management of bookings, properties, and users. Admins can add new bookings, properties, and manage users via modals. Each tab includes interactive elements like search, filters, and clickable actions to view, edit, or delete items. The modal system is used for adding or editing items or confirming deletions. The layout is responsive and includes icons and modern UI elements to enhance user experience.

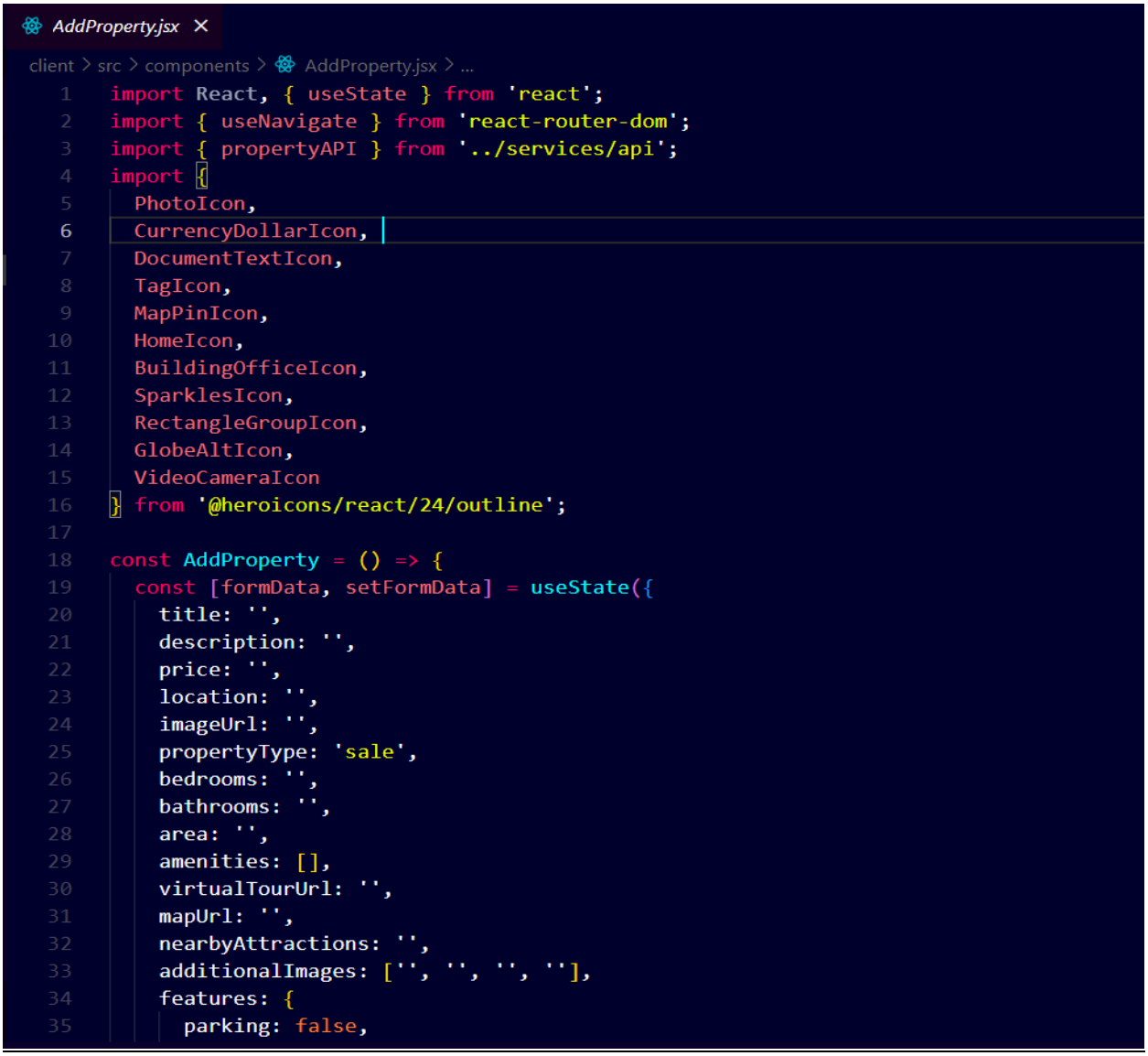
About.jsx



Explanation:

The About page of the Horizon Homes website provides an overview of the company, its values, team, milestones, and achievements. The page includes a hero section with a brief introduction, followed by several sections that highlight key statistics (such as years of experience, happy clients, properties sold, and expert agents). It also features the company's core values, including client-focused service, integrity, excellence, and innovation. The Team Section introduces key team members with their roles, bios, and specialties. The Milestones Section outlines significant events in the company's history, showcasing its growth over the years. Additionally, the page has a Call to Action encouraging visitors to contact the company or browse available properties. The page uses smooth animations for each section to create a dynamic and engaging user experience.

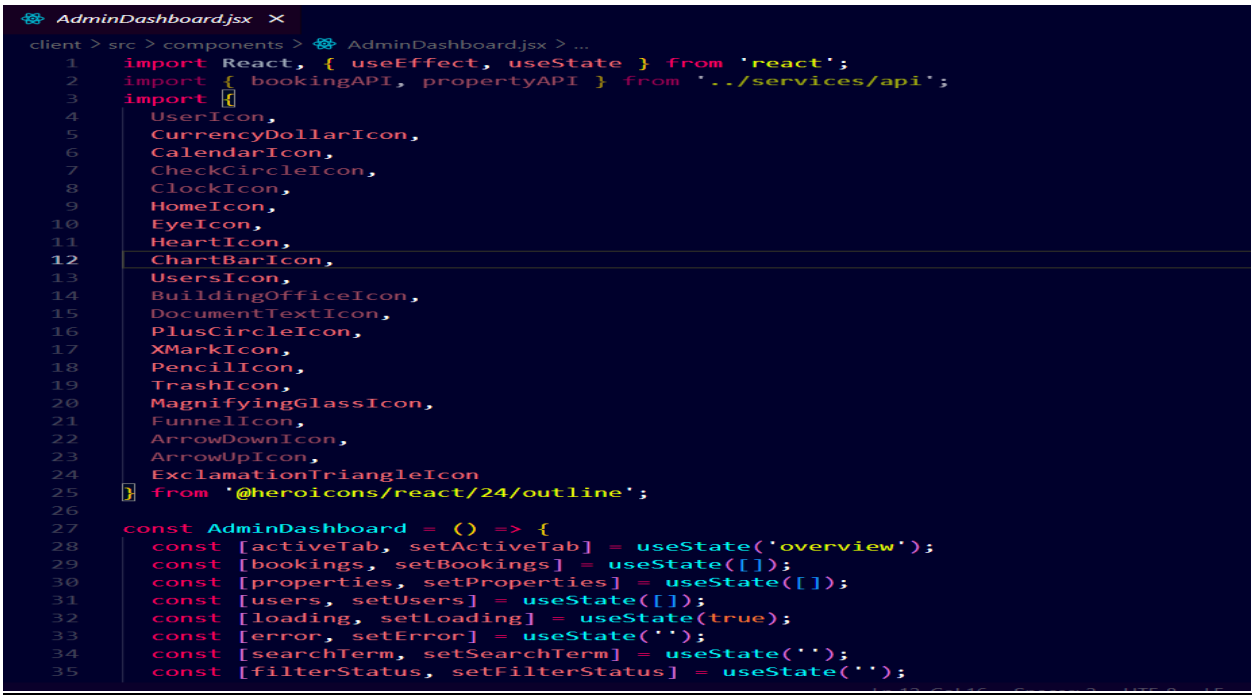
AddProperty.jsx



Explanation:

The `AddProperty` React component is designed to manage the creation of a new property listing. It uses the `useState` hook to manage form data, such as the property title, description, price, location, image URLs, features, and amenities. These state values are updated through a `handleChange` function, which listens for changes in the form fields, including text inputs, checkboxes, and file uploads. The component includes a form with various fields for entering property details like price, location, number of bedrooms, bathrooms, and property features. Users can also upload images and preview them before submitting the form. The form submission is handled by the `handleSubmit` function, which validates the data, checks that required fields are filled in, and ensures the price is a positive number. If validation passes, the form sends the property data to a backend API for storage. If the submission is successful, the user is redirected to the dashboard. For UI features, the form uses Tailwind CSS for styling and icons to visually enhance the input fields, providing a better user experience. Error messages are displayed if there are any issues with the submission. A loading spinner appears during the submission process to indicate progress. Additionally, the component conditionally renders previews of the main property image and any additional images that the user has selected.

AdminDashboard.jsx

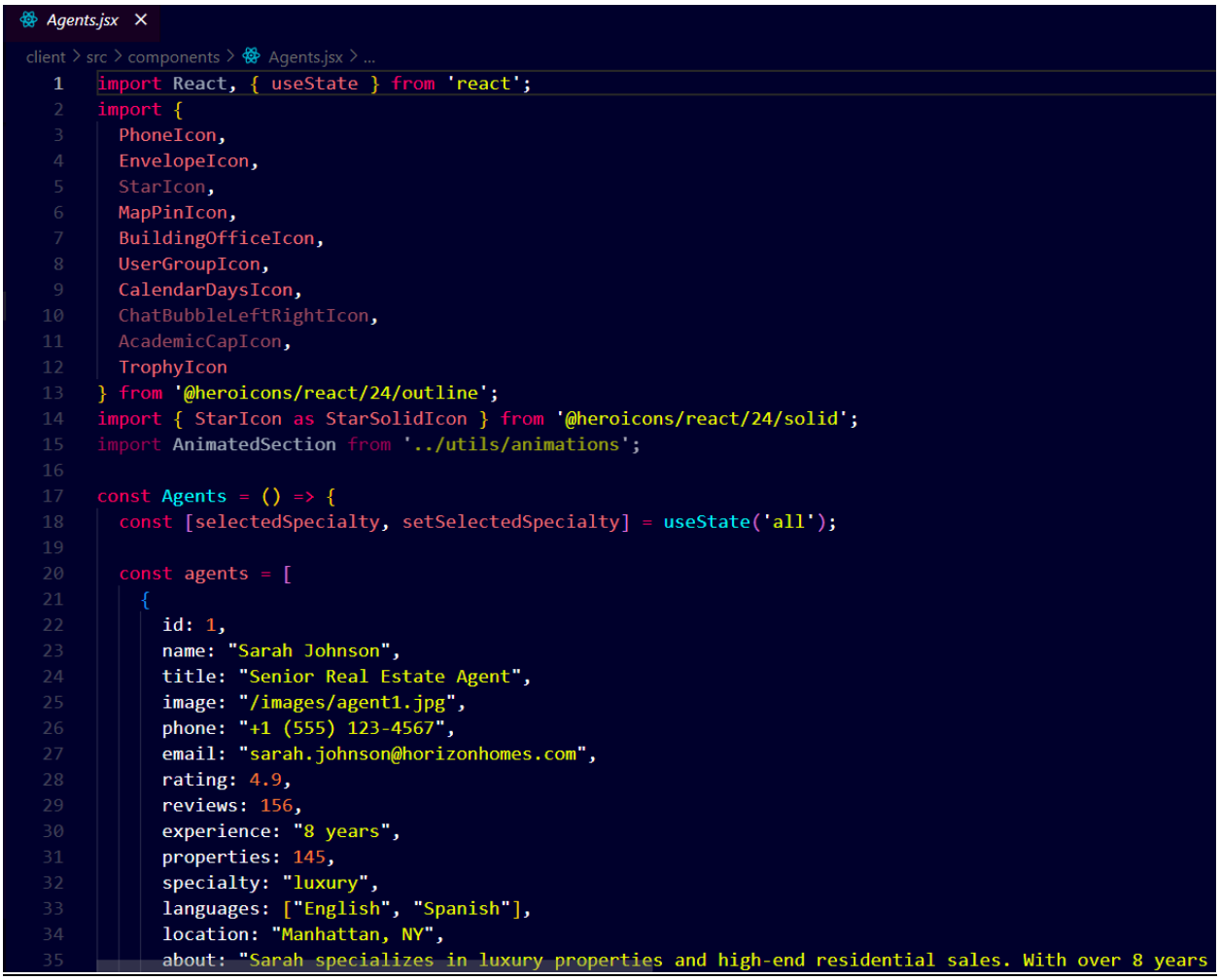


Explanation:

The `AdminDashboard` component is a central hub for managing the real estate platform. It allows administrators to monitor and control bookings, properties, and users. Upon loading, the dashboard fetches sample data for bookings, properties, and users, and presents an overview with key statistics like the total number of bookings, active properties, total revenue, and active users.

The dashboard has several sections, each represented by tabs: Overview, Bookings, Properties, and Users. In the Overview tab, admins can quickly see metrics like the total bookings, active properties, and total revenue. The Bookings tab allows admins to view, search, and filter through bookings, update their statuses, and delete them. In the Properties tab, admins can manage property listings, including adding new properties, editing, and deleting existing ones. The Users tab displays a list of platform users, showing their details and allowing for deletion of user accounts. The component uses a modal dialog for actions like adding or editing properties, bookings, and users, or confirming deletion. The modal adjusts its content based on the action being performed. The component also includes error handling and loading indicators. During data fetching or form submission, a loading spinner is displayed, and if any errors occur, a message is shown to the user. To enhance the user experience, the admin can filter and search through the bookings, properties, and users based on different parameters. Currency values for properties are formatted in USD for easy reading. The component is built using React hooks for state management and employs `@heroicons` for displaying icons. It also uses routing to navigate between different sections of the admin dashboard.

Agents.jsx



Explanation:

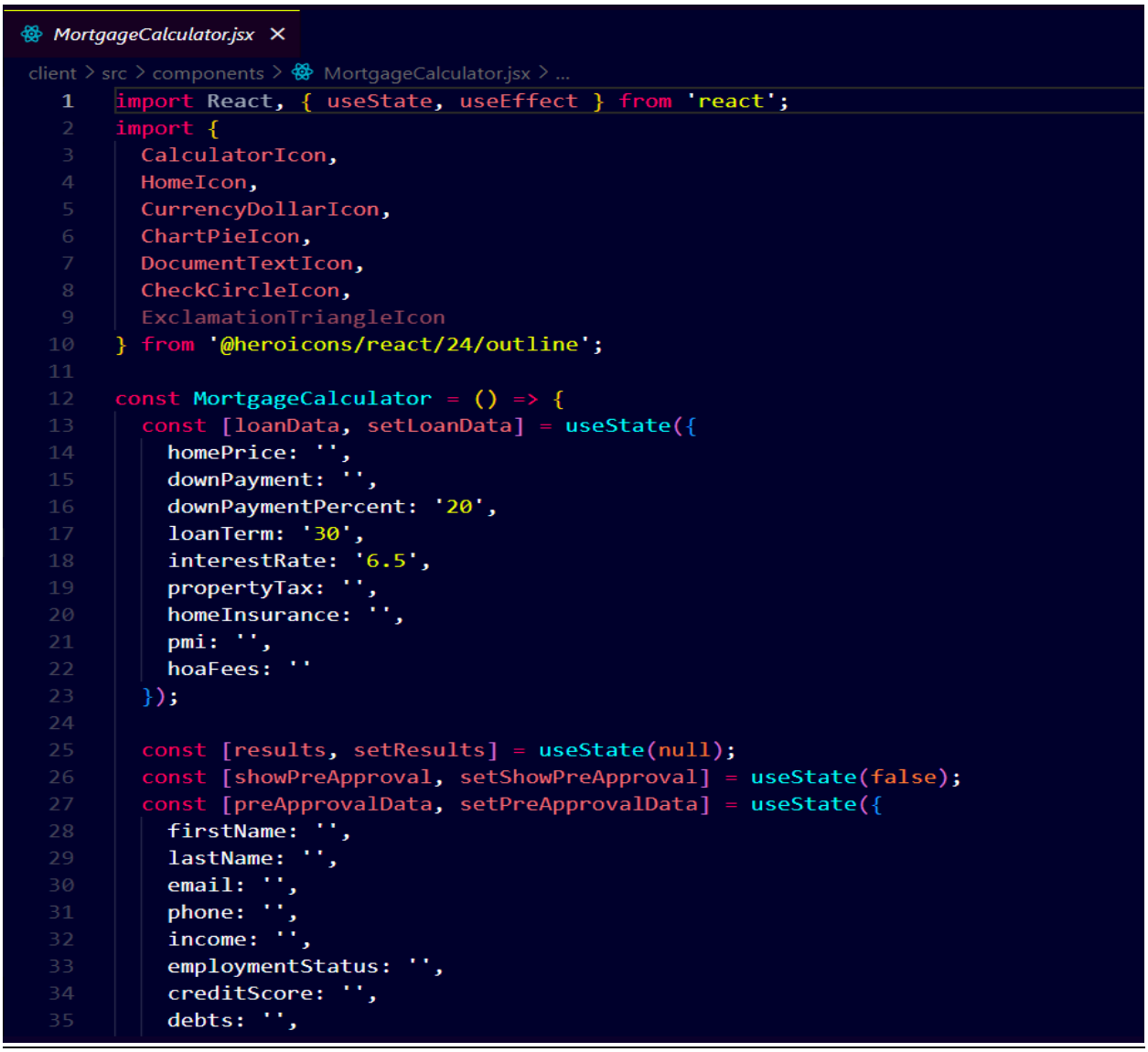
The `Agents` component is a part of the Horizon Homes real estate platform, designed to showcase the team's real estate professionals. This component is structured to display a list of agents along with their specialties, experience, ratings, and contact details. Users can filter agents based on their specialty (such as luxury properties, commercial real estate, residential properties, or investment properties). The component includes a hero section at the top with an engaging introduction, followed by a filter section where users can choose agents by specialty. Each agent's profile card displays important details such as their photo, title, experience, location, and a brief about their expertise. The profile also includes contact buttons for calling or emailing the agent, along with a scheduling option for meetings. Ratings and reviews are prominently displayed to highlight the agents' reputation and experience. The agents' specialties and certifications are also listed, allowing users to easily identify their expertise in various property types. The grid of agent profiles is dynamic, allowing users to browse through different agents and their specialties. The layout includes interactive elements like buttons to contact agents directly via phone or email, and to schedule a consultation. At the bottom, there's a call-to-action encouraging users to connect with the agents or schedule a consultation with the team. The component uses state management to handle the selected specialty filter and dynamically adjusts the displayed agents based on the user's selection. Additionally, there are animations and hover effects on agent cards, creating a smooth and engaging user experience. The overall design is responsive, ensuring that the component looks great on both mobile and desktop devices.

Blog.jsx

Explanation:

The `Blog` component is a blog page for the Horizon Homes website, showcasing real estate-related articles. The page has a clean, engaging design that includes a hero section with a heading and a brief description. It features a search bar and category filter options for users to easily find articles of interest. The content is dynamically displayed using a grid layout for blog posts, each showing the article's title, author, publish date, and excerpt. Featured posts are highlighted to draw attention, and users can view full articles with a button click. The blog also includes filtering by categories such as "Market Insights," "Buying Guides," "Selling Tips," and more. Each blog post card shows the article's image, tags, and a button to read more. Additionally, a newsletter signup section encourages visitors to stay updated with the latest articles. The page is built with animations, smooth transitions, and responsive design to ensure a pleasant browsing experience across all devices.

MortageCalculator.jsx



Explanation:

The `MortgageCalculator` component is a comprehensive tool designed to help users calculate their monthly mortgage payments based on input parameters such as home price, down payment, loan term, interest rate, and additional costs like property tax, insurance, PMI, and HOA fees. The component dynamically updates the results as the user modifies any input fields, providing a breakdown of the total monthly payment and the loan summary. The component also includes a pre-approval application form, allowing users to submit personal and financial information to get pre-approved for a mortgage. The results and pre-approval form are styled with clean, responsive UI elements, using icons for enhanced user experience. Additionally, current market interest rates are displayed to help users make informed decisions. The page includes animations and interactive elements for a smooth user experience.

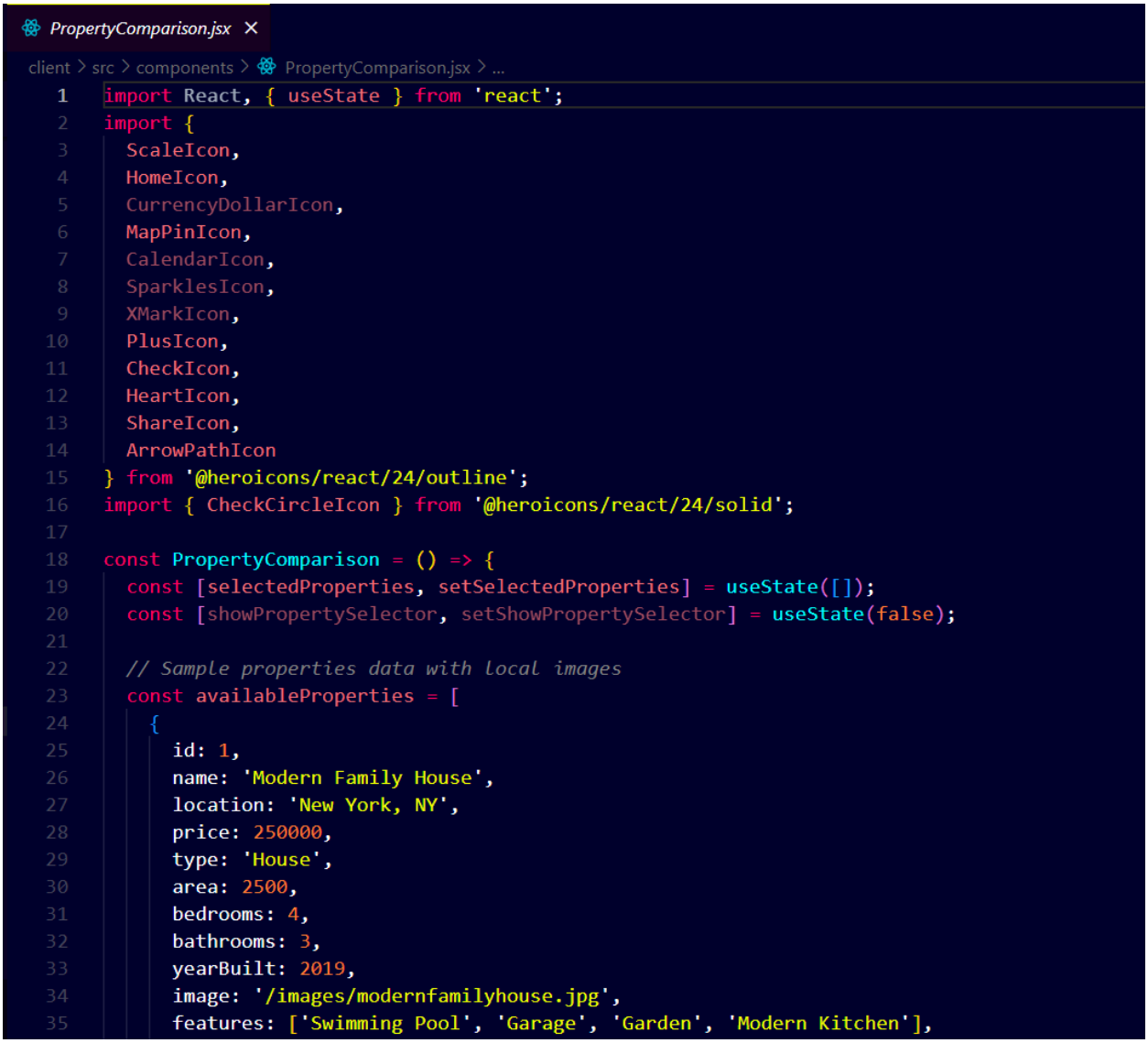
MyBooking.jsx

Explanation:

The `MyBookings` component fetches and displays a list of property bookings made by the logged-in user. It manages state for bookings, loading, and error messages using React's `useState` and `useEffect` hooks. Upon mounting, it calls the `fetchBookings` function to fetch data using the `bookingAPI.getMyBookings()` method. If the data is successfully retrieved, the list of bookings is displayed. Each booking is shown with the property details, status (pending or completed), price, and the seller's information. If no bookings are found, a message prompting the user to browse properties is shown. The component also handles loading states with a spinning loader and error

states by displaying an alert message. The bookings are rendered as cards with images and status badges. The user can click on the "Browse Properties" button to explore available properties.

PropertyComparison.jsx

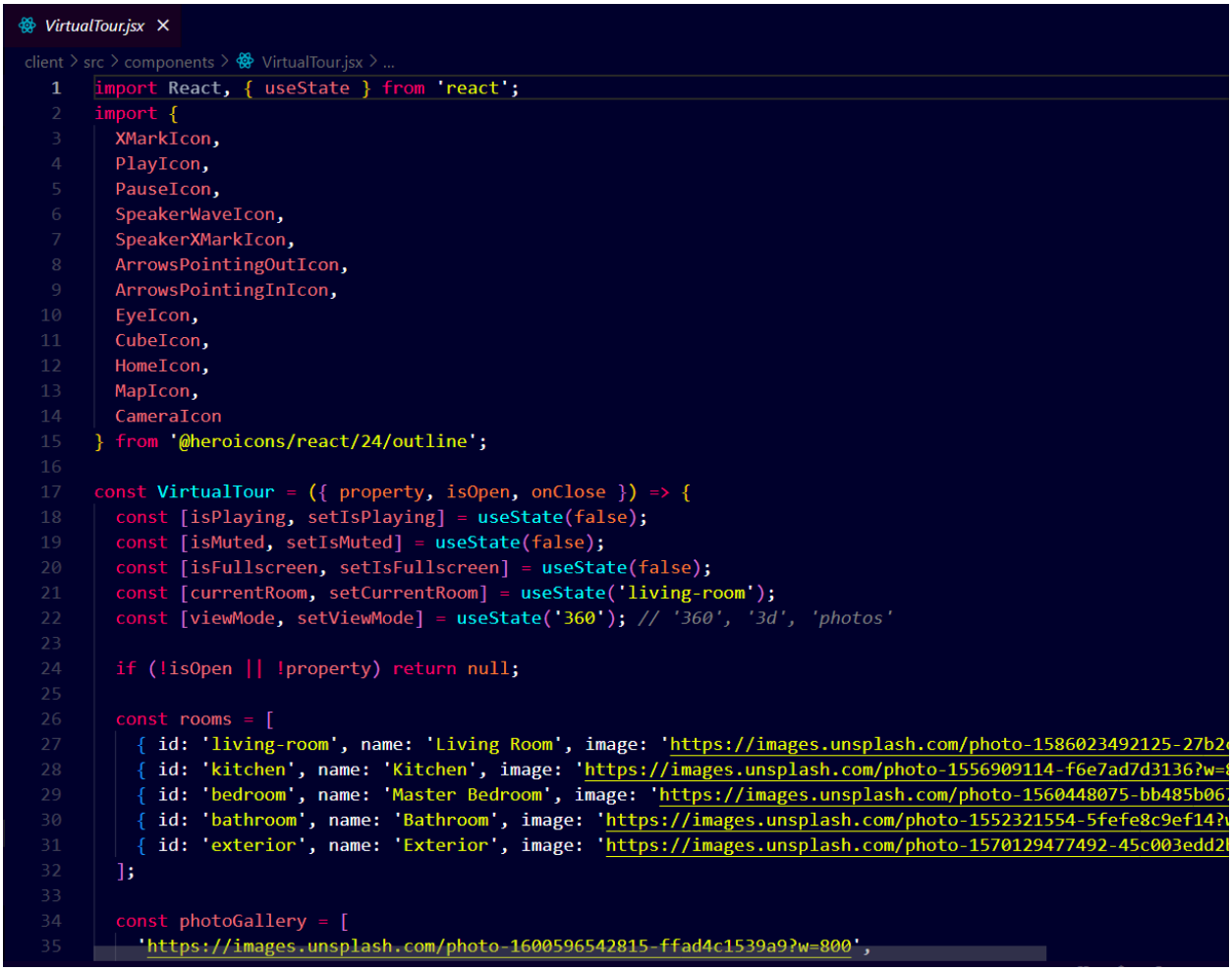


Explanation:

The `PropertyComparison` component allows users to compare up to three properties side by side. It manages selected properties and handles displaying a list of available properties that can be added to the comparison. The properties are shown with their images, prices, features, and descriptions. Users can add or remove properties from their selection, with a limit of three properties at a time. If properties are selected, a detailed comparison table is shown, with each property's key attributes displayed side by side, such as price, area, features, and more. Users can also share or reset their comparison. The component includes a simple error handling mechanism,

and if no properties are selected, it shows an empty state message. The properties data is hardcoded in the component for the sake of simplicity, and sample images are used for each property.

VirualTour.jsx

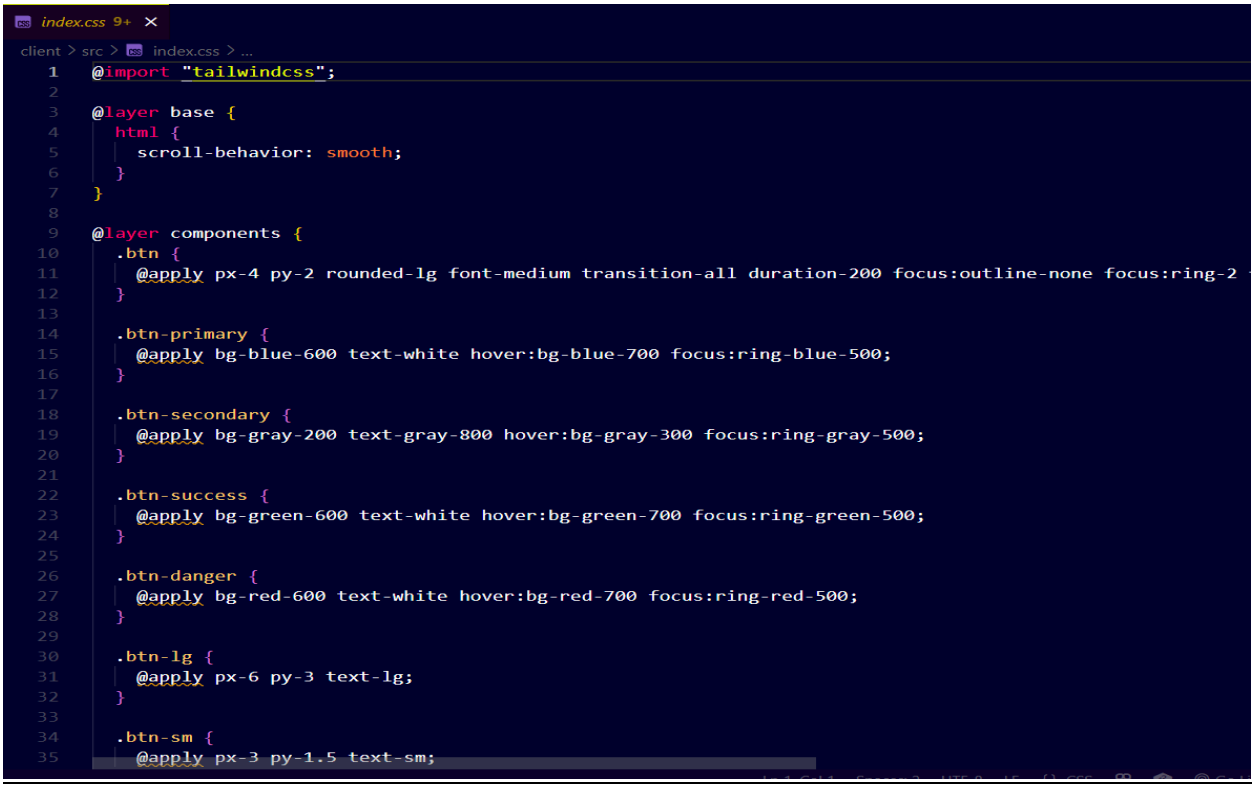


```jsx
import React, { useState } from 'react';
import {
  XMarkIcon,
  PlayIcon,
  PauseIcon,
  SpeakerWaveIcon,
  SpeakerXMarkIcon,
  ArrowsPointingOutIcon,
  ArrowsPointingInIcon,
  EyeIcon,
  CubeIcon,
  HomeIcon,
  MapIcon,
  CameraIcon
} from '@heroicons/react/24/outline';

const VirtualTour = ({ property, isOpen, onClose }) => {
  const [isPlaying, setIsPlaying] = useState(false);
  const [isMuted, setIsMuted] = useState(false);
  const [isFullscreen, setIsFullscreen] = useState(false);
  const [currentRoom, setCurrentRoom] = useState('living-room');
  const [viewMode, setViewMode] = useState('360'); // '360', '3d', 'photos'

  if (!isOpen || !property) return null;

  const rooms = [
    { id: 'living-room', name: 'Living Room', image: 'https://images.unsplash.com/photo-1586023492125-27b2
    { id: 'kitchen', name: 'Kitchen', image: 'https://images.unsplash.com/photo-1556909114-f6e7ad7d3136?w=
    { id: 'bedroom', name: 'Master Bedroom', image: 'https://images.unsplash.com/photo-1560448075-bb485b06
    { id: 'bathroom', name: 'Bathroom', image: 'https://images.unsplash.com/photo-1552321554-5fefe8c9ef14?
    { id: 'exterior', name: 'Exterior', image: 'https://images.unsplash.com/photo-1570129477492-45c003edd2
  ];

  const photoGallery = [
    'https://images.unsplash.com/photo-1600596542815-ffad4c1539a9?w=800',
```

Explanation:

The `VirtualTour` component provides an interactive virtual tour experience for a property. It features various view modes like 360° view, 3D model, and photo gallery. The component manages different states for playing, muting, and fullscreen toggling, allowing users to control the tour experience. It has a sidebar for navigating between rooms, displaying details like bedrooms, bathrooms, square footage, and property type. Users can interact with images and switch between views using buttons for each mode. The control bar at the bottom includes play/pause, mute, and fullscreen options, as well as navigation for viewing the floor plan. The property details, such as the name and location, are shown at the top. The component also supports closing the tour and toggling between rooms and media.

index.css



Explanation:

This is a custom Tailwind CSS configuration that enhances the default styling with utility classes and animations. It defines a variety of button styles like `.btn-primary`, `.btn-secondary`, and `.btn-success` for different button types, which apply consistent padding, border-radius, hover effects, and focus states. There are also form input styles like `.form-input`, `.form-textarea`, and `.form-select`, which are styled for consistent appearance and focus states across the form elements. Additionally, custom card styles like `.card` and `.property-card` offer hover transitions and shadows for interactive visual effects. Alerts, with different types like `.alert-success`, `.alert-error`, and `.alert-warning`, are also included for consistent notifications. For animations, custom keyframes are defined such as `float`, `pulse-glow`, and `shimmer`, which are applied to create visually engaging effects like glowing buttons or shimmering backgrounds. The `.glass-morphism` class implements a glass-like background effect using `backdrop-filter`, creating a frosted glass effect. To enhance the text appearance, the `.gradient-text` class applies a gradient to the text, and `.input-focus-effect` provides a subtle transform and shadow effect when input fields are focused. In terms of interactivity, `.btn-fancy` and `.card-lift` introduce hover effects, such as animated background gradients for buttons and lifted transformations with shadow effects for cards. Lastly, responsive design helpers like `.mobile-padding` ensure proper padding for smaller screen sizes (max-width: 640px), improving mobile usability.

App.jsx

```jsx
import UserProfile from './components/UserProfile';
import Contact from './components/Contact';
import About from './components/About';
import Blog from './components/Blog';
import Services from './components/Services';
import Agents from './components/Agents';
import Reviews from './components/Reviews';
import PropertyDetails from './components/PropertyDetails';
import PropertyComparison from './components/PropertyComparison';

// Protected Route component
const ProtectedRoute = ({ children }) => {
  const { isAuthenticated, loading } = useAuth();

  if (loading) {
    return (
      <div className="flex items-center justify-center min-h-screen">
        <div className="animate-spin rounded-full h-12 w-12 border-b-2 border-blue-600"></div>
      </div>/.flex.items-center.justify-center.min-h-screen
    );
  }

  return isAuthenticated ? children : <Navigate to="/login" />;
};

// Admin Route component
const AdminRoute = ({ children }) => {
  const { isAuthenticated, isAdmin, loading } = useAuth();

  if (loading) {
    return (
      <div className="flex items-center justify-center min-h-screen">
        <div className="animate-spin rounded-full h-12 w-12 border-b-2 border-blue-600"></div>
      </div>/.flex.items-center.justify-center.min-h-screen
```

Explanation:

This code is for the main structure of a React application for a real estate platform. Here's a breakdown of its components:

1. Routing:
    - The app uses React Router (`BrowserRouter`, `Routes`, `Route`, and `Navigate`) to handle navigation between pages.
    - The `Router` wraps the entire app to enable routing functionalities.
    - Various routes are defined for different pages, such as `Home`, `Login`, `Register`, `Dashboard`, `Properties`, `PropertyDetails`, `Agents`, and more.
2. Protected and Admin Routes:
    - ProtectedRoute: A higher-order component that ensures only authenticated users can access certain routes (like `Dashboard`, `AddProperty`, and `MyBookings`). If the user is not authenticated, they are redirected to the `Login` page.
    - AdminRoute: This component ensures that only users with admin rights can access the admin-related routes (like `AdminDashboard`). If the user isn't an admin, they are redirected to their dashboard.
3. AuthContext:
    - The AuthProvider wraps the app to provide authentication-related context (like `isAuthenticated`, `isAdmin`, and `loading`) to all components.
    - The `useAuth` hook is used within protected and admin routes to check if the user is authenticated and has the right role.
4. Components:
    - Navbar: A navigation bar component, typically containing links to different sections of the website.

- Other components like `Dashboard`, `Properties`, `Login`, `Register`, `UserProfile`, `AdminDashboard`, `MortgageCalculator`, `FAQ`, etc., represent various pages of the app.
- Routes are defined to display these components based on the URL path.

In summary, this is a React app that manages user authentication and role-based access (admin or regular user) while providing navigation between various sections of the platform like properties, bookings, and user profiles.

- Backend:

server.js

```
server.js
backend > server.js > ...
1   const express = require('express');
2   const mongoose = require('mongoose');
3   const cors = require('cors');
4   require('dotenv').config();
5
6   // Import routes
7   const authRoutes = require('./routes/auth');
8   const propertyRoutes = require('./routes/properties');
9   const bookingRoutes = require('./routes/bookings');
10
11  const app = express();
12
13  // Middleware
14  app.use(cors());
15  app.use(express.json());
16
17  // Routes
18  app.use('/api/auth', authRoutes);
19  app.use('/api/properties', propertyRoutes);
20  app.use('/api/bookings', bookingRoutes);
21
22  // Health check endpoint
23  app.get('/api/health', (req, res) => {
24    res.json({ message: 'Server is running successfully!' });
25  });
26
27  // Error handling middleware
28  app.use((err, req, res, next) => {
29    console.error(err.stack);
30    res.status(500).json({ message: 'Something went wrong!' });
31  });
32
33  // 404 handler
34  app.use('*', (req, res) => {
35    res.status(404).json({ message: 'Route not found' });
```

Explanation:

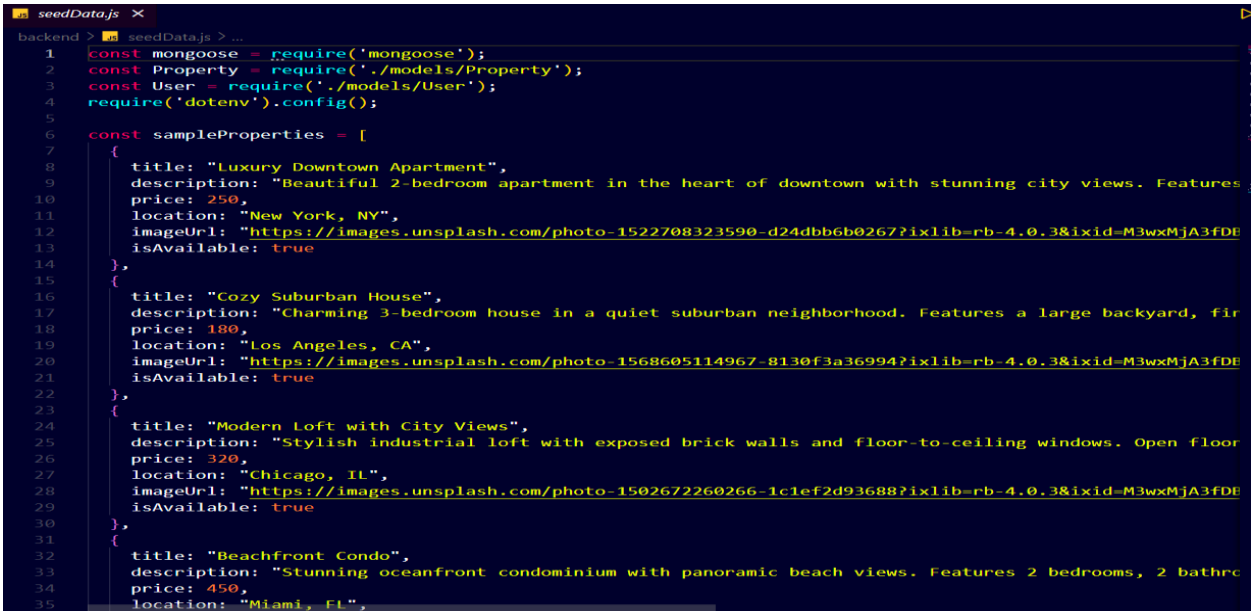
This code sets up an Express.js server for a real estate application, with the following key functionalities:

1. Dependencies:
   - Express for the web server.
   - Mongoose for MongoDB integration.
   - CORS to allow cross-origin requests.
   - dotenv to manage environment variables, like database URI.

2. Middleware:
    - CORS and express.json() middleware are used to handle cross-origin requests and parse incoming JSON data, respectively.
3. Routes:
    - Routes for authentication (`/api/auth`), property management (`/api/properties`), and booking management (`/api/bookings`) are imported and added to the app.
    - A health check endpoint (`/api/health`) is provided to verify that the server is running properly.
4. Error Handling:
    - Global error handler: Catches unexpected errors and responds with a 500 status code.
    - 404 handler: Responds with a 404 status code for any undefined routes.
5. MongoDB Connection:
    - The app connects to MongoDB using Mongoose, with the connection string stored in environment variables.
    - If the connection is successful, the server starts listening on a specified port (`5000` by default). If it fails, the process exits with an error.

This setup is the foundation for the backend of the real estate application, where different routes handle user authentication, property management, and booking operations.

seeddata.js



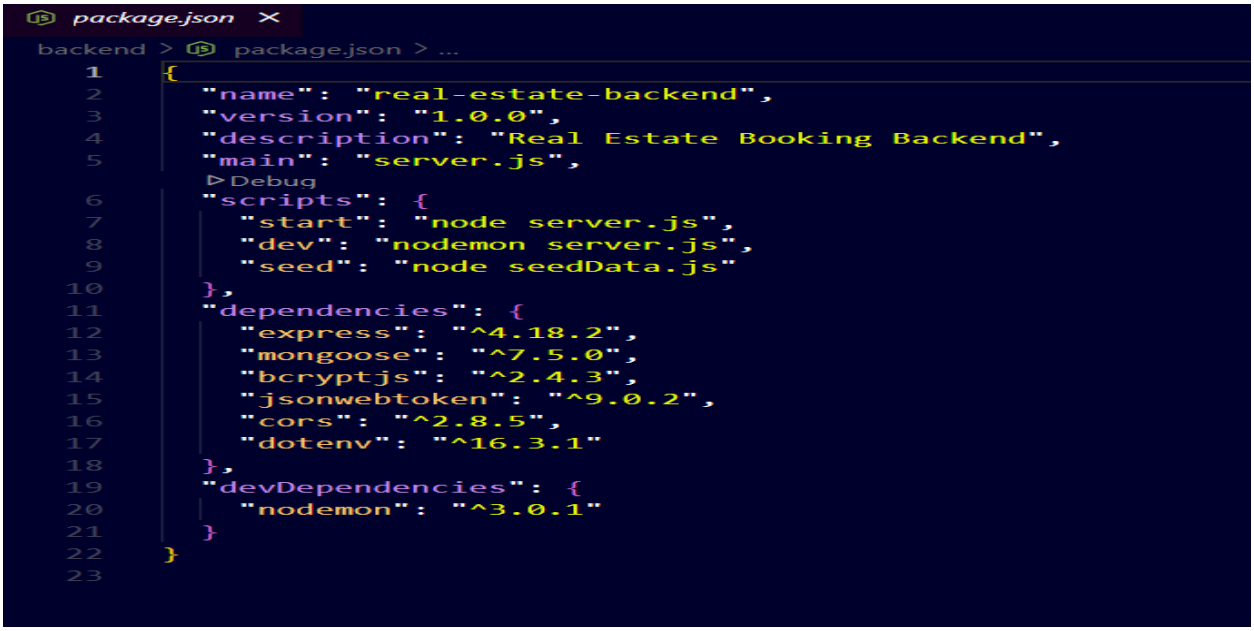
Explanation:

This script is used to seed a MongoDB database with sample real estate property data. Here's a breakdown of its steps:

1. Import dependencies:
    - `mongoose` for interacting with MongoDB.
    - `User` and `Property` models to interact with the corresponding collections.
    - `dotenv` to load environment variables like the MongoDB connection URI.
2. Sample Properties Data: A set of sample property data (`sampleProperties`) is defined, each with details like title, description, price, location, and an image URL.
3. Seed Function (`seedDatabase`):
    - Connect to MongoDB: It establishes a connection to the database using the URI stored in the `.env` file.
    - Create a Demo User: A demo user (property owner) is checked in the `User` collection by email. If not found, a new user is created with a predefined email and placeholder password.
    - Clear Existing Properties: Optionally clears any existing properties from the database.
    - Add Properties: The sample properties are then added to the database, each associated with the demo user as the owner.
4. Exit Process: After the seeding is done, the script exits the process with `process.exit(0)`.

This script is typically used for populating a new database with initial data for testing or development purposes.

package.json



Explanation:

This is a `package.json` file for a Node.js project named "real-estate-backend". Here's a breakdown of the key sections:

1. Metadata:
    - name: The name of the project, "real-estate-backend".
    - version: The current version of the project, "1.0.0".
    - description: A brief description of the project, "Real Estate Booking Backend".
    - main: The entry point of the application, `server.js`.
2. Scripts:
    - start: Runs the application using `node server.js`.
    - dev: Starts the application with `nodemon`, which automatically restarts the server on code changes.
    - seed: Runs a script called `seedData.js` (likely to populate the database with initial data).
3. Dependencies:
   These are libraries required to run the project:
    - express: Web framework for building the API.
    - mongoose: MongoDB ODM to interact with MongoDB.
    - bcryptjs: Used for hashing passwords.
    - jsonwebtoken: To create and verify JSON Web Tokens for authentication.
    - cors: Middleware to handle Cross-Origin Resource Sharing (CORS).
    - dotenv: Loads environment variables from a `.env` file.
4. DevDependencies:
    - nodemon: Development tool that automatically restarts the server on file changes.

This file sets up the environment for developing and running the real estate backend application.

Booking.js

Booking.js

backend > models > Booking.js > ...

```
const mongoose = require('mongoose');

const bookingSchema = new mongoose.Schema({
  property: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Property',
    required: true
  },
  buyer: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true
  },
  seller: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true
  },
  price: {
    type: Number,
    required: true
  },
  status: {
    type: String,
    enum: ['pending', 'completed', 'cancelled'],
    default: 'completed'
  }
}, {
  timestamps: true
});

module.exports = mongoose.model('Booking', bookingSchema);

```

Explanation:

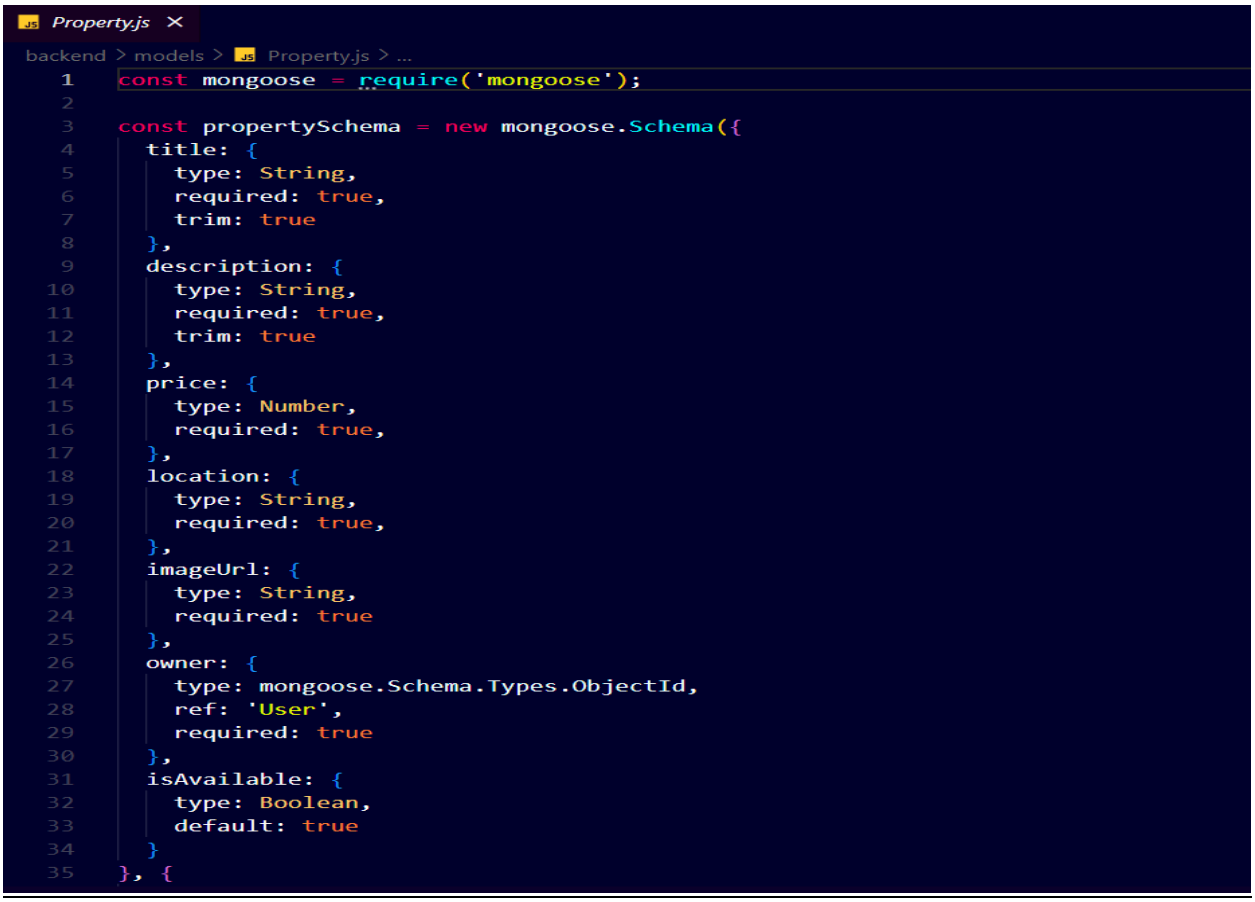
This code defines a Booking model using Mongoose for MongoDB in a Node.js application, which is used to store booking information for properties. The schema includes the following fields:

1. property: A reference to the Property model (using the property ObjectId), representing the property that is being booked. It is required.
2. buyer: A reference to the User model (using the user's ObjectId), representing the buyer (or renter) who made the booking. It is required.
3. seller: A reference to the User model (using the user's ObjectId), representing the seller (or landlord) involved in the booking. It is required.
4. price: A required number that indicates the price of the property in the booking.
5. status: A string that can be one of the following values: 'pending', 'completed', or 'cancelled'. It defaults to 'completed'.

The model also includes timestamps, automatically adding `createdAt` and `updatedAt` fields.

This schema helps manage the booking process for properties, tracking who is involved, the price, and the status of each booking.

Property.js



Explanation:

This code defines a Property model using Mongoose for MongoDB in a Node.js application, which will be used to store property-related data in the database. The model includes the following fields:

1. title: A required string that represents the title of the property.
2. description: A required string that provides a detailed description of the property.
3. price: A required number that indicates the price of the property.
4. location: A required string specifying the property's location.
5. imageUrl: A required string that stores the URL of the property's image.
6. owner: A reference to the User model (using the user's ObjectId), representing the owner of the property.
7. isAvailable: A boolean indicating if the property is available for sale or rent, defaulting to `true`.

The model also includes timestamps, automatically adding `createdAt` and `updatedAt` fields to track when the property is created or modified.

This schema is used to represent properties listed for sale or rent, allowing users to manage property details in the database.

User.js

Explanation:

This code defines a User model using Mongoose for MongoDB in a Node.js application. The model includes the following fields:

1. name: A required string representing the user's name.
2. email: A required, unique string for the user's email (it is also converted to lowercase and trimmed).
3. password: A required string with a minimum length of 6 characters, storing the user's hashed password.
4. role: A string that can be either 'user' or 'admin', with a default value of 'user'.

Features:

- Pre-save Hook: Before saving a user, the password is hashed using bcryptjs if it is modified, ensuring that the password is securely stored.
- comparePassword Method: A method to compare a candidate password with the stored hashed password for login authentication.

This model provides user authentication functionality and handles secure password storage and comparison.

auth.js

Explanation:

This code defines two middleware functions for handling authentication and authorization in a Node.js application using JWT tokens:

1. auth: This middleware is responsible for verifying if the incoming request has a valid JWT token. It extracts the token from the `Authorization` header, decodes it using the secret key (`process.env.JWT_SECRET`), and checks if the user exists in the database. If the token is missing or invalid, or if the user isn't found, it sends an error response. If everything is valid, it adds the user data to the `req.user` object and proceeds to the next middleware.
2. adminAuth: This middleware first invokes the `auth` function to authenticate the user, then checks if the authenticated user has an `admin` role. If the user is not an admin, it returns a `403 Forbidden` error. If the user is authenticated and an admin, it allows the request to continue.

These middlewares ensure that only authenticated users (and admins for specific routes) can access certain resources.

7. Screenshots of Web Pages / API Testing:

HOME PAGE:

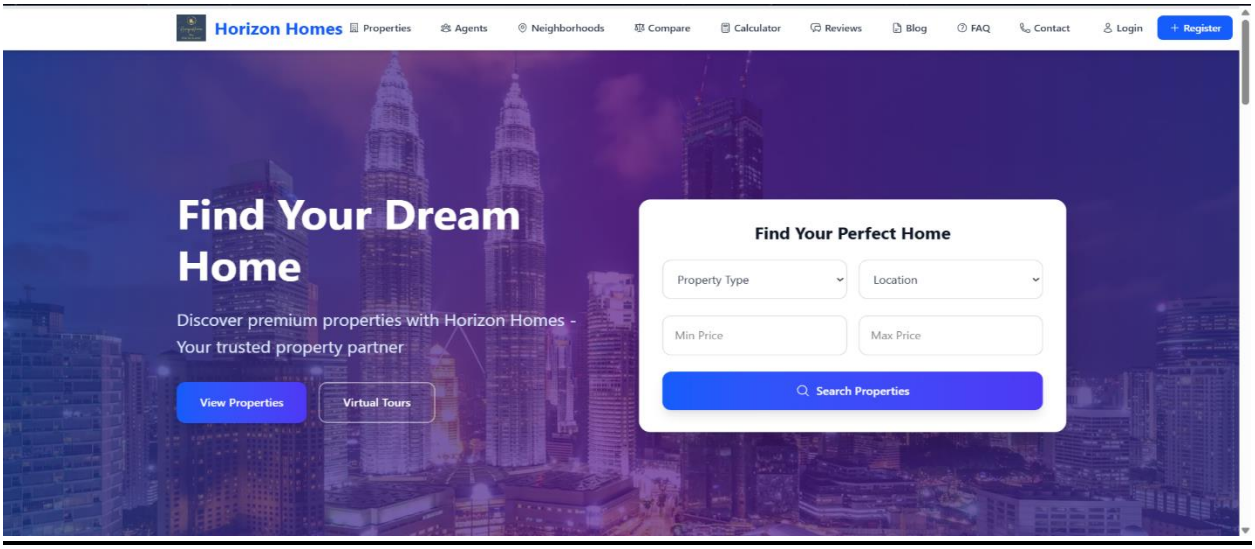


Figure 01: Home Page

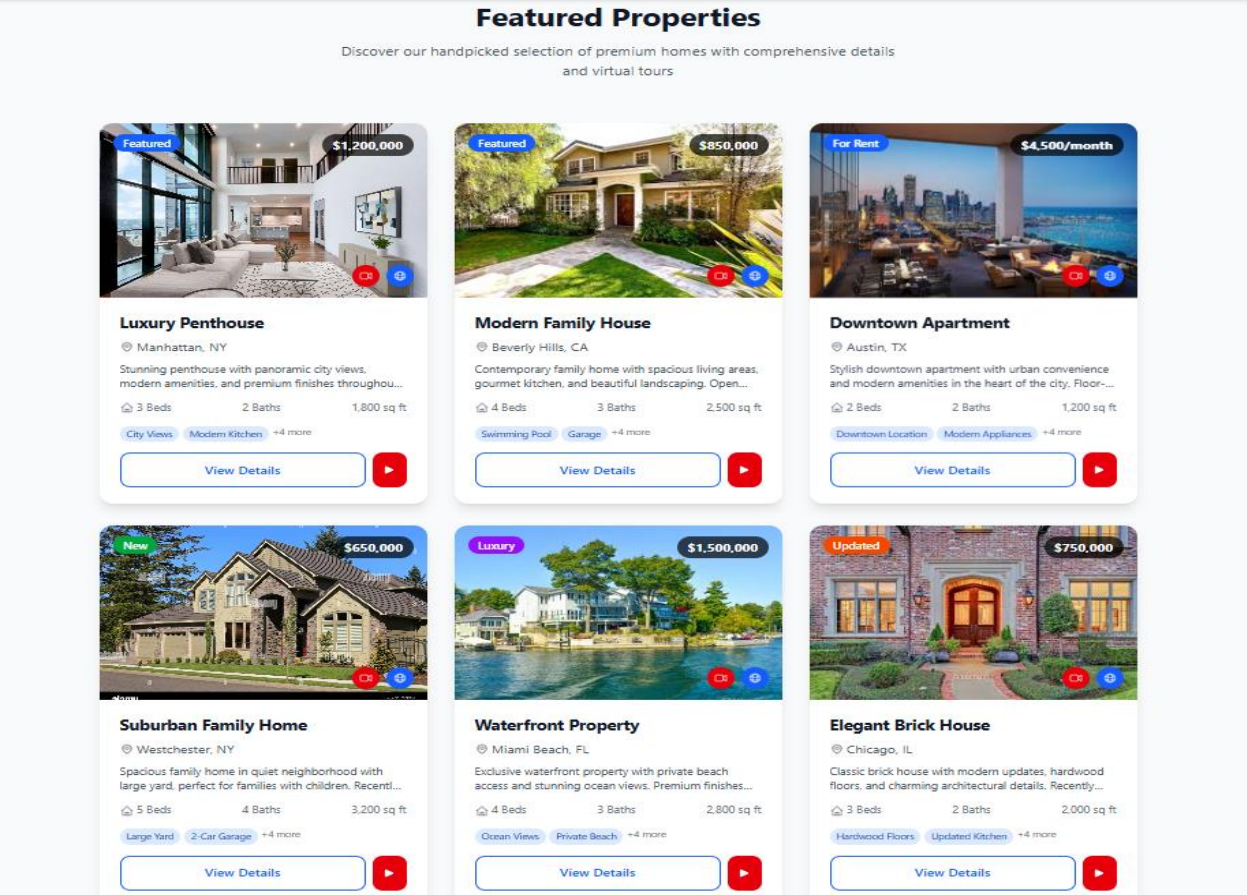


Figure 02: Home Page

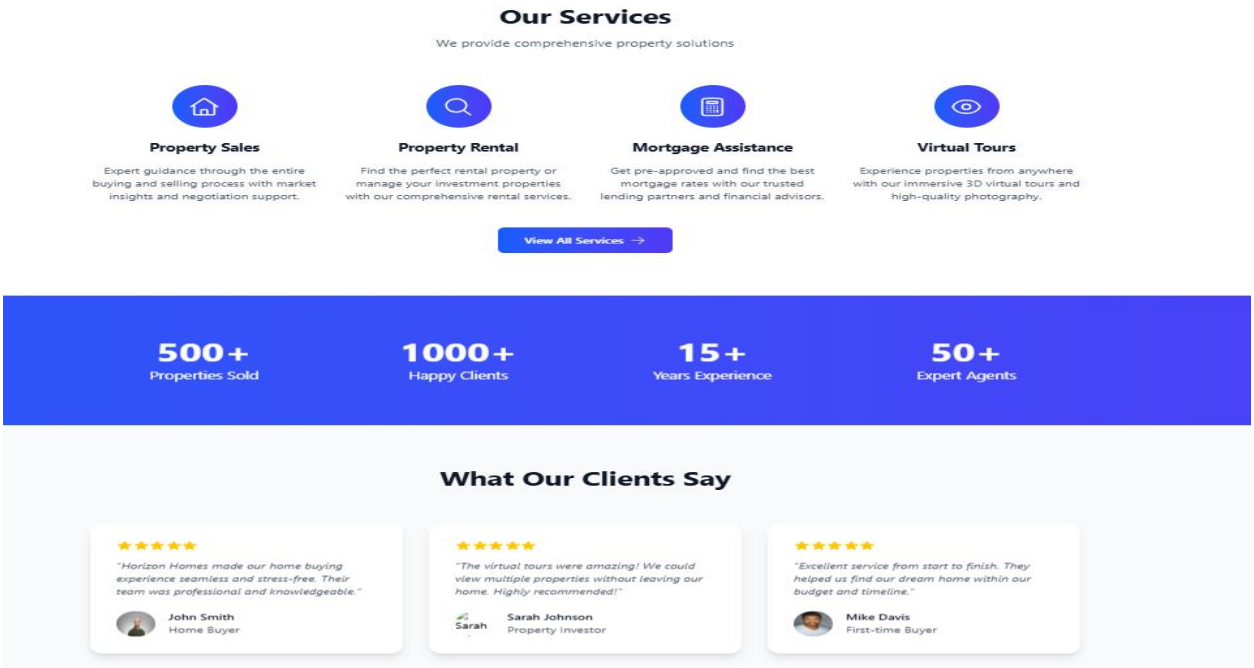


Figure 03: Home Page

LOGIN OR SIGN UP:

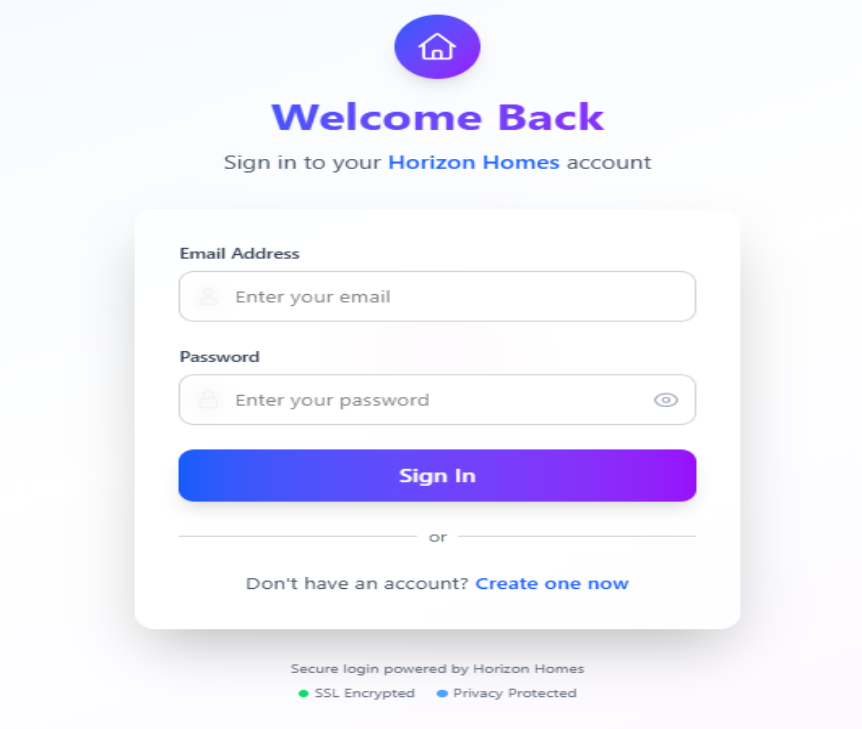


Figure 04: Login page

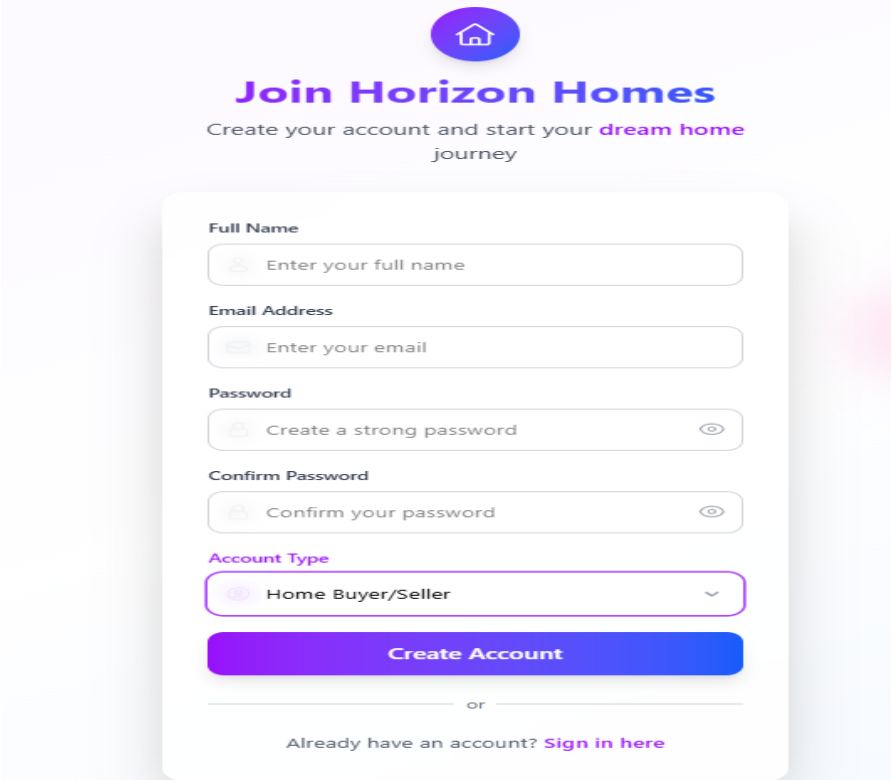


Figure 05: SignUp page

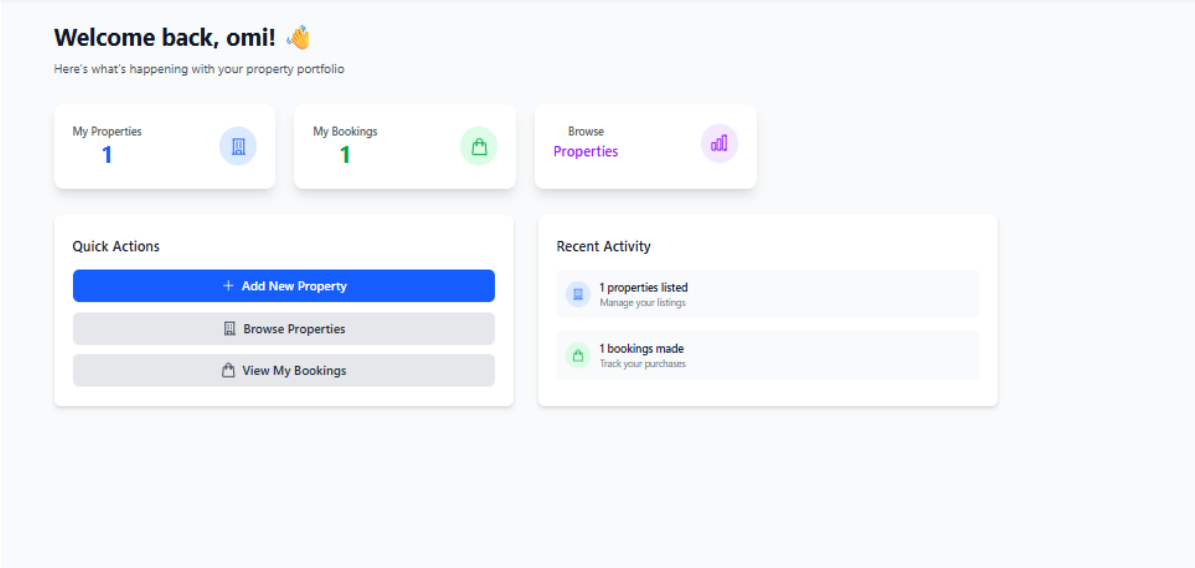


Figure 06: After Login Dashboard

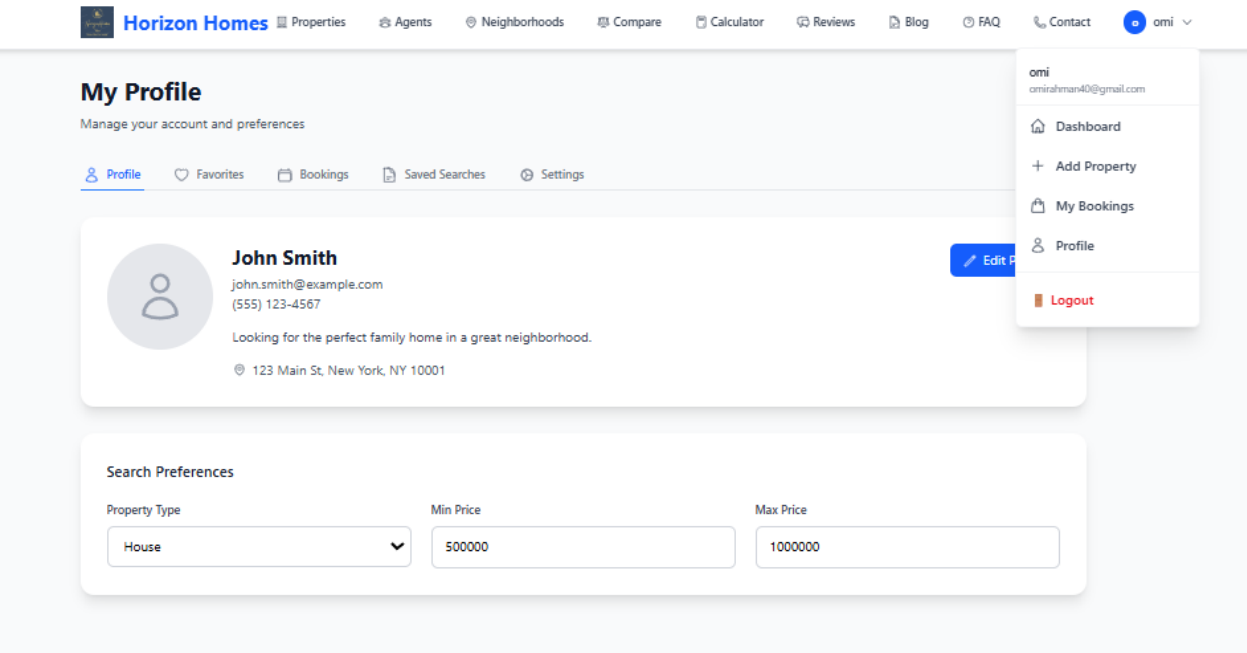


Figure 07: After Login Can Edit Their Profile

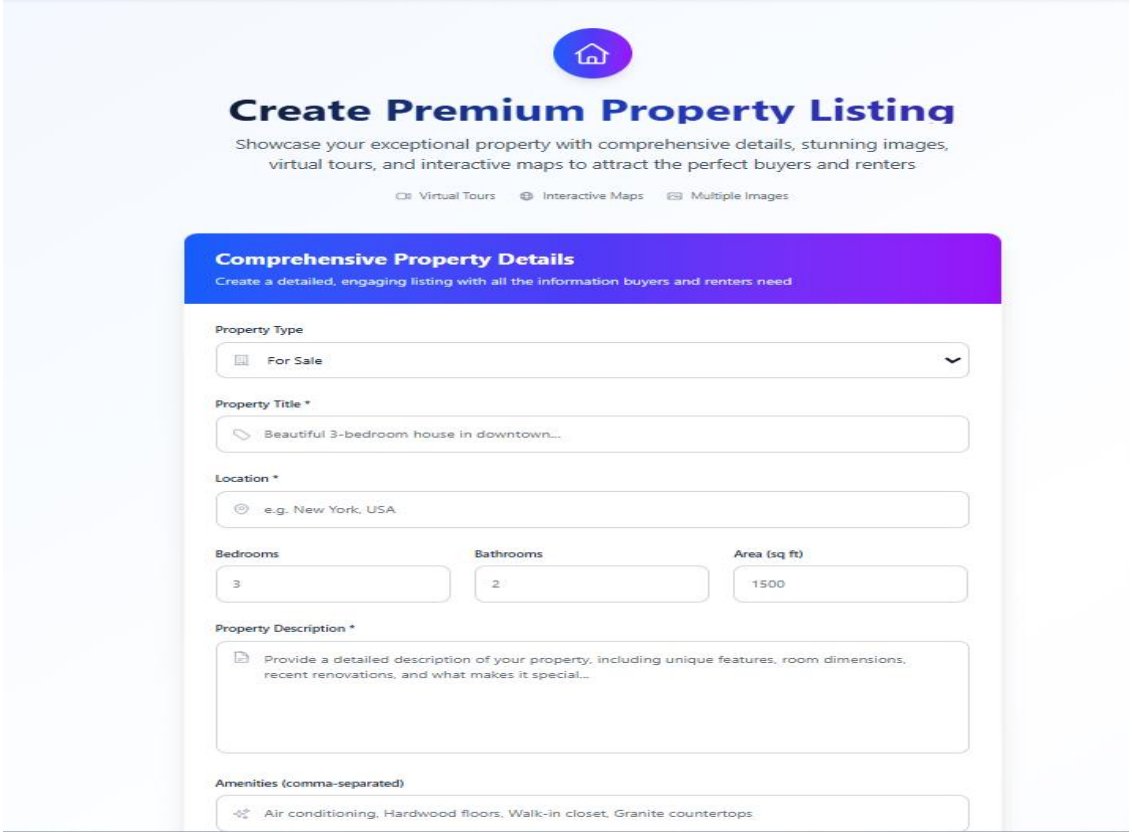


Figure 08: Property Adding By Users

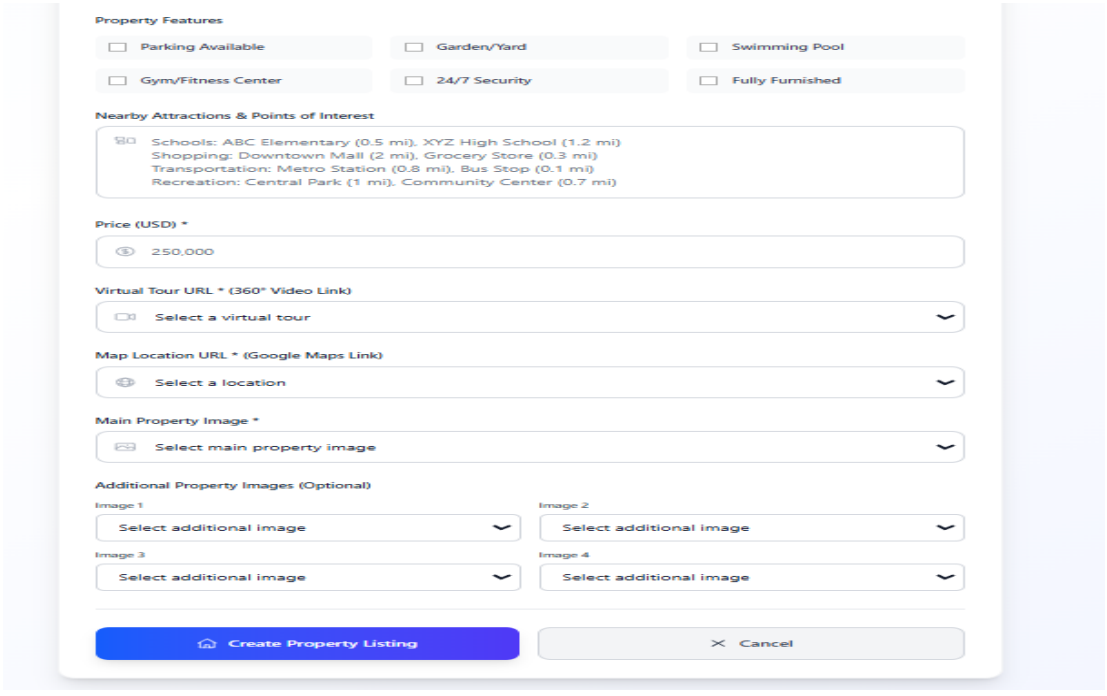


Figure 09: Property Adding By Users

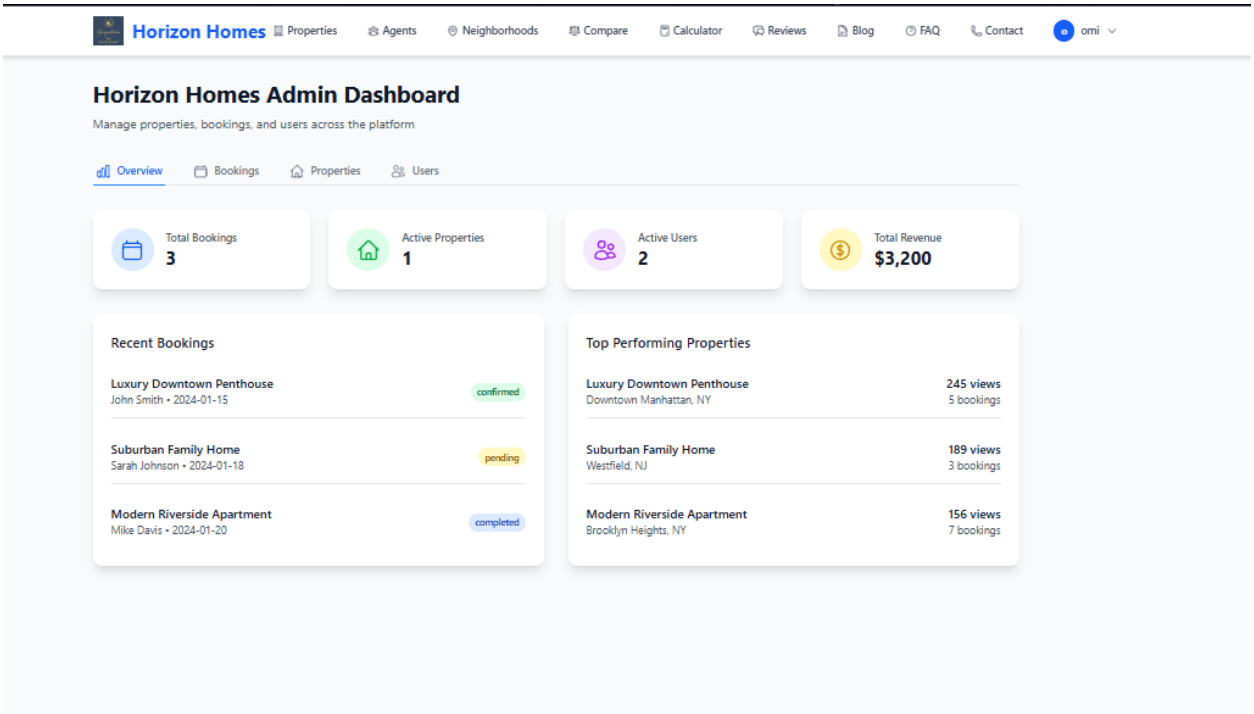


Figure 10: Admin dashboard

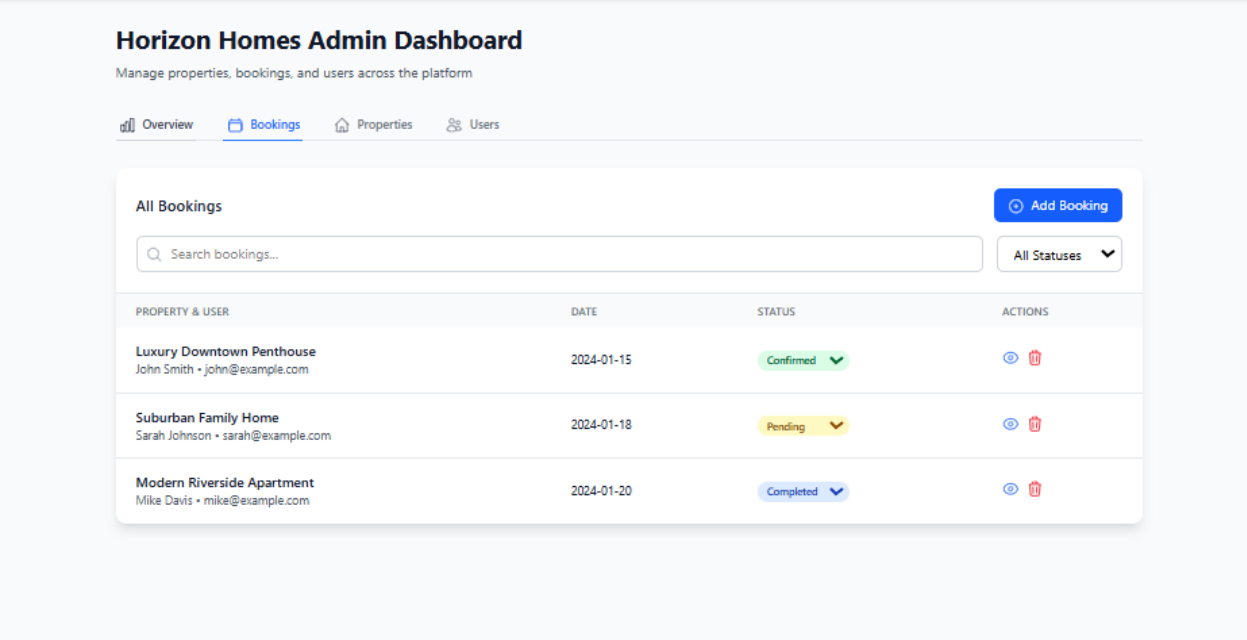


Figure 11: Admin dashboard (Booking)

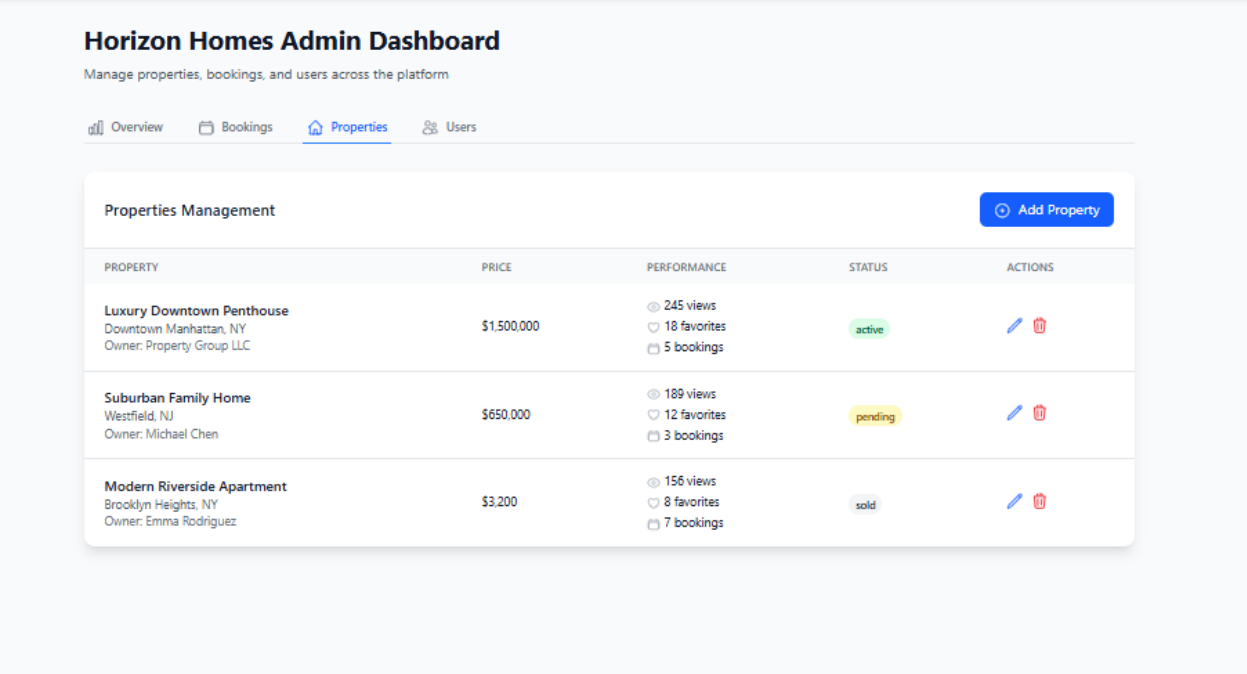


Figure 12: Admin dashboard (Properties)

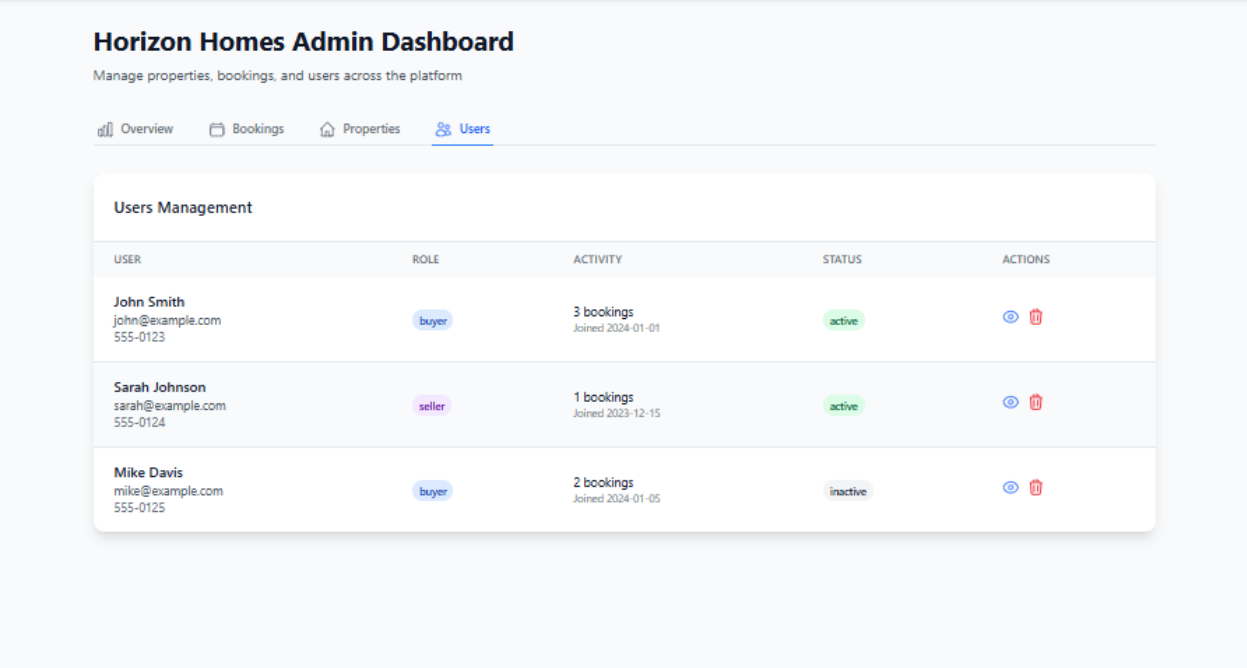


Figure 13: Admin dashboard (Users)

FEATURED PROPERTIES:

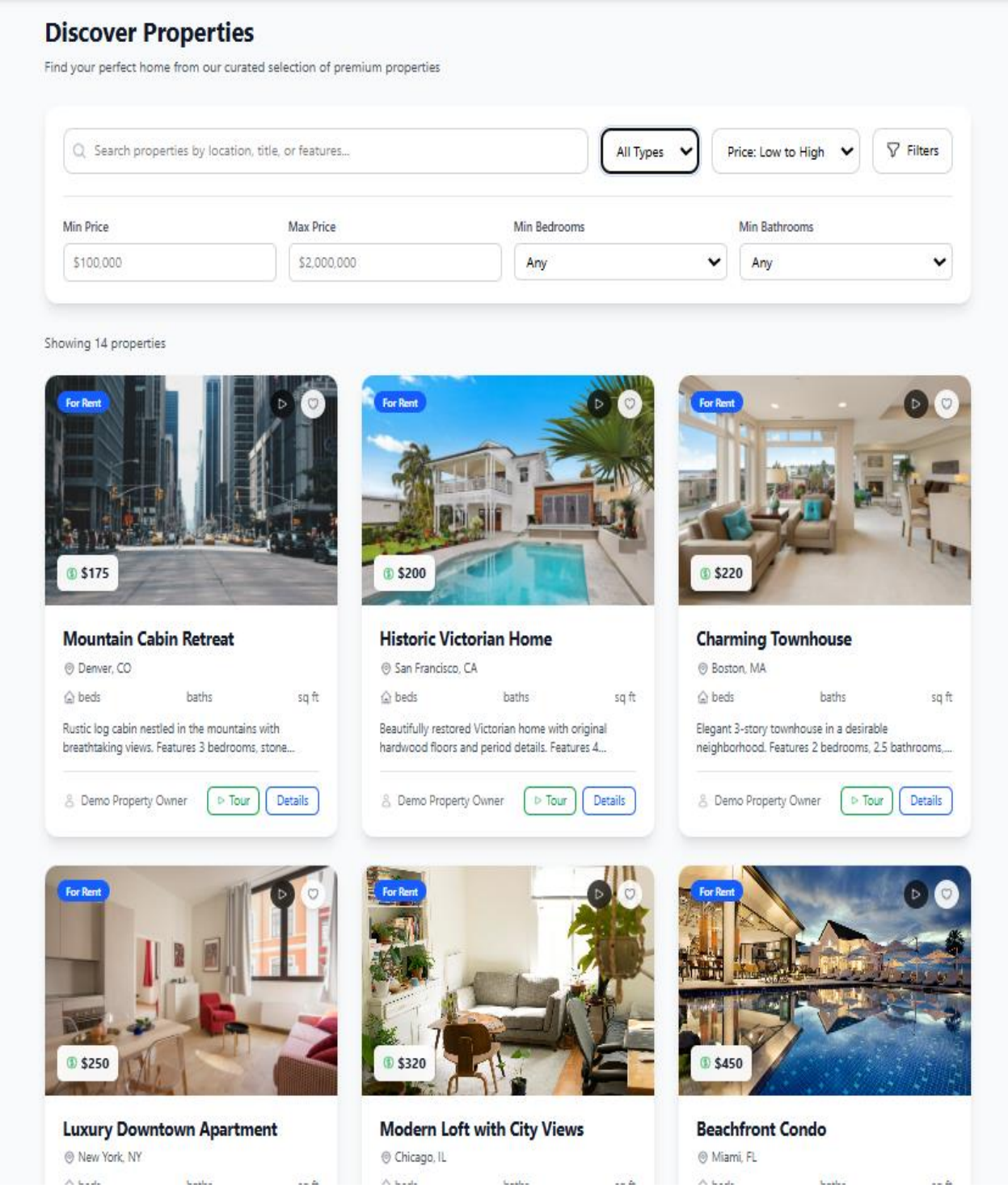


Figure 14: Property Listing Page

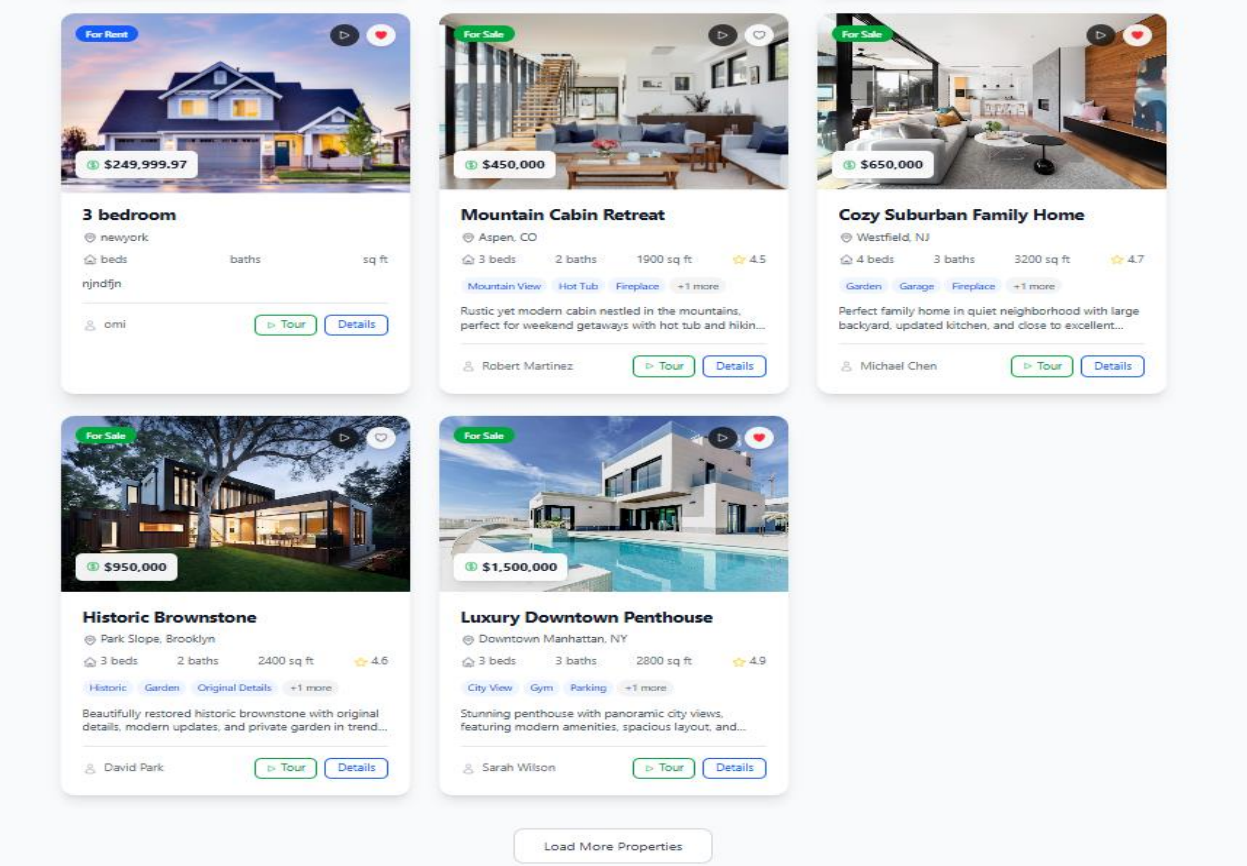


Figure 15: Property Listing Page (Mark As favourites)

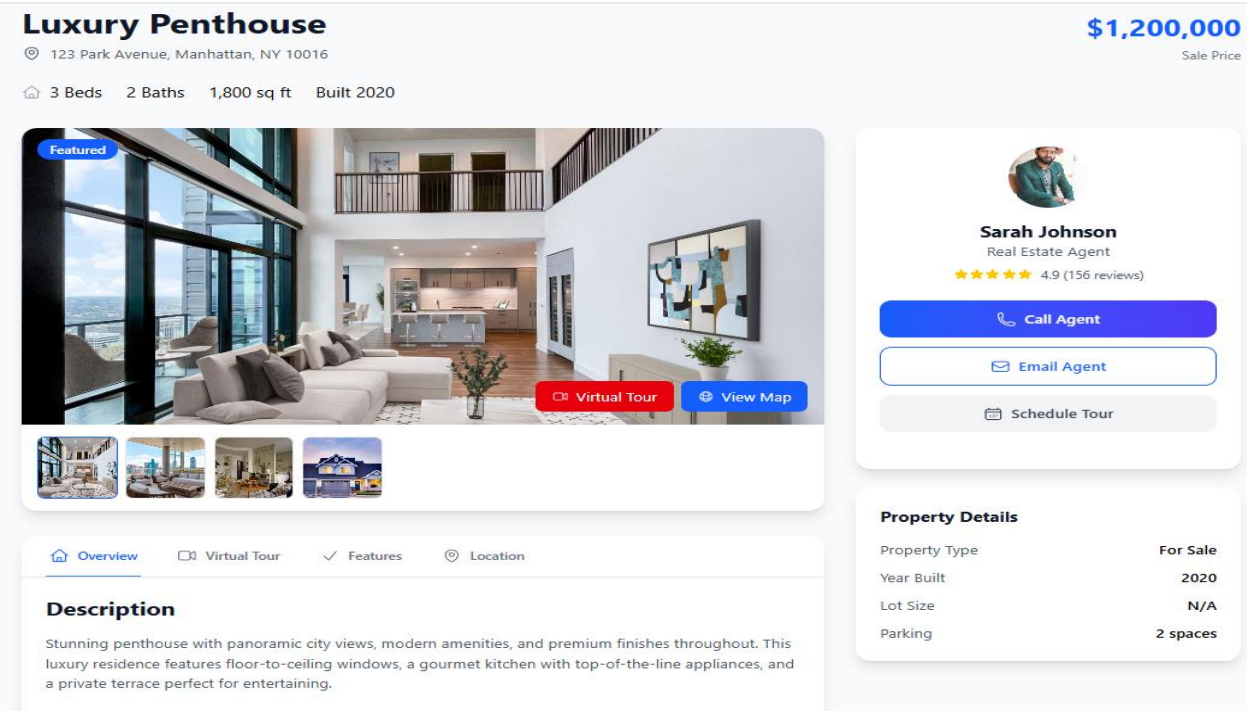


Figure 16: Property Description Page

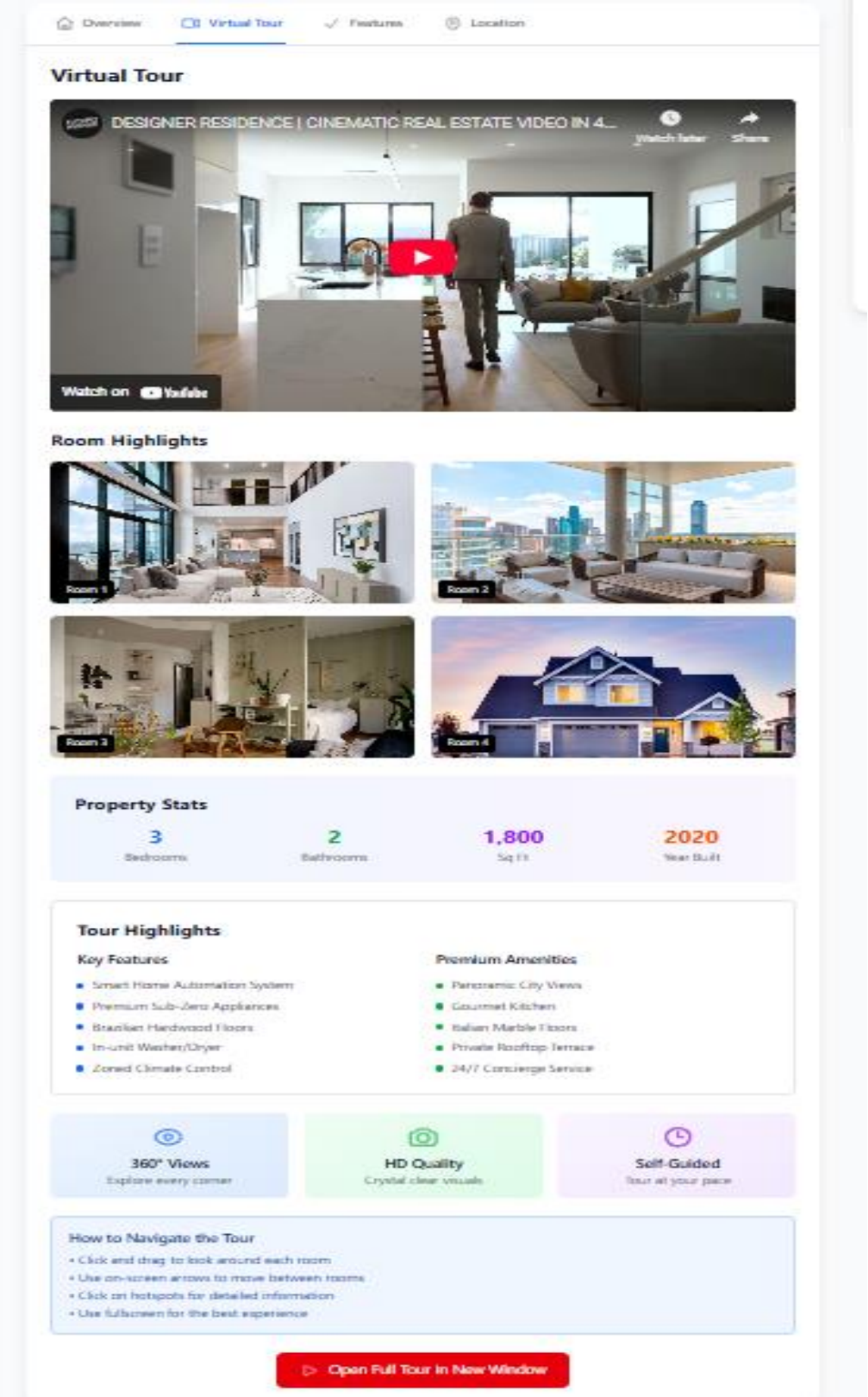


Figure 17: Property Description Page (Virtual Tour)

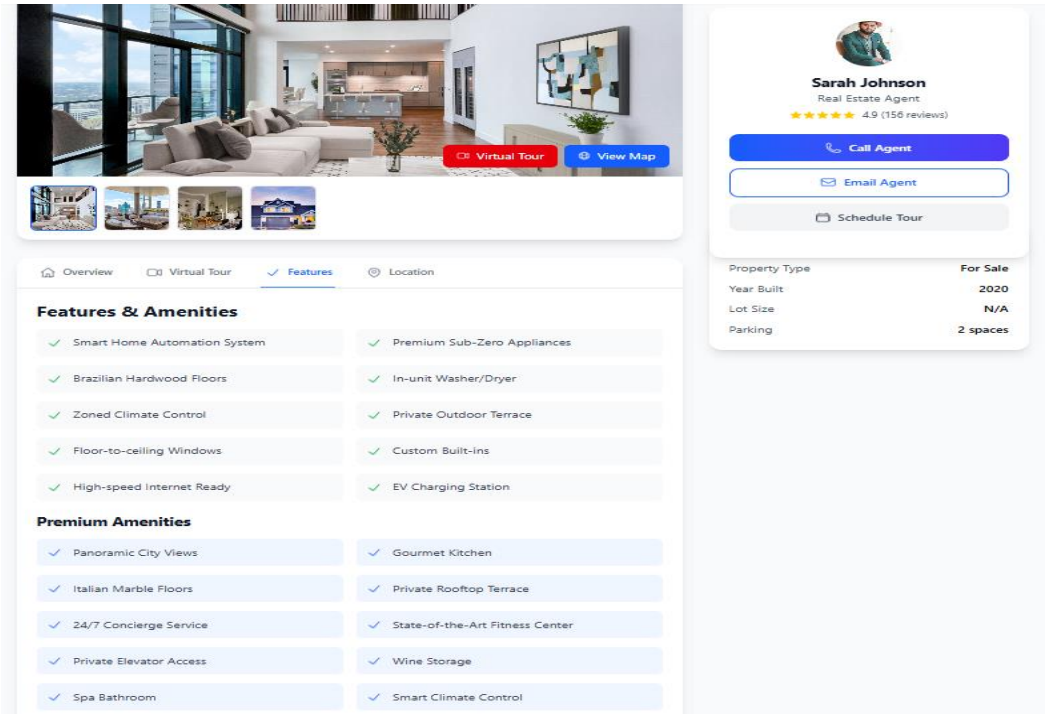


Figure 18: Property Description Page (Particular Key Features of Properties)

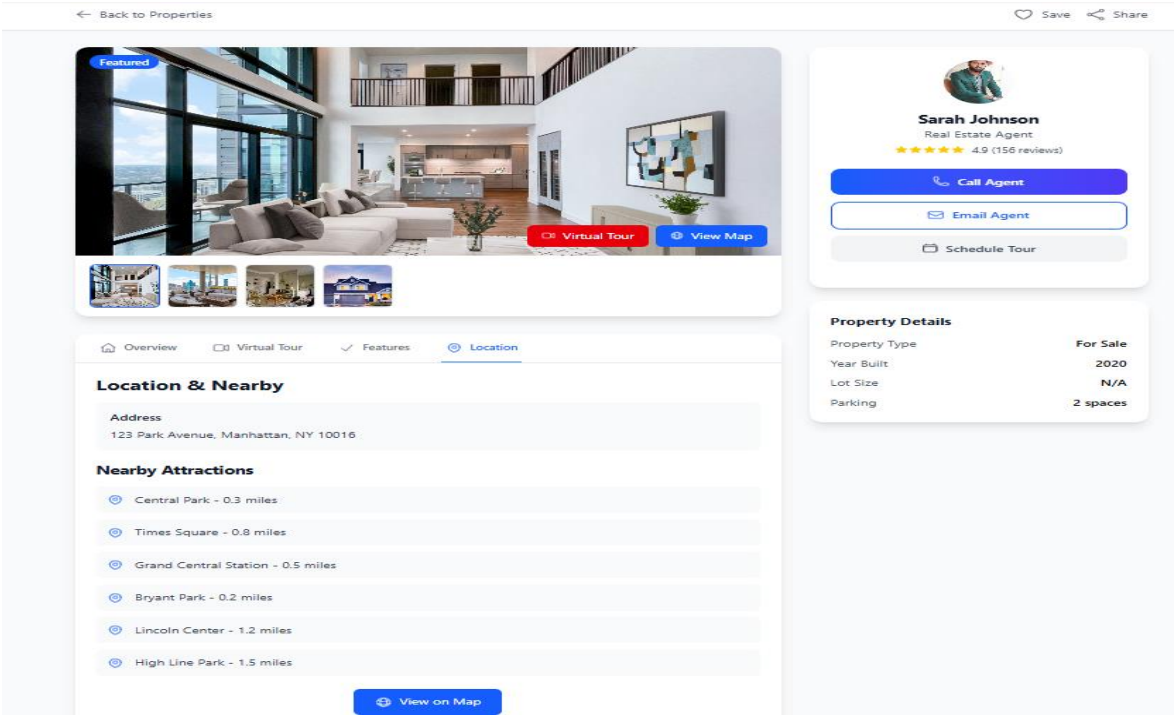


Figure 19: Property Description Page (Location of Properties on Map)

AGENT PAGE:

Figure 20: Agent Description Page

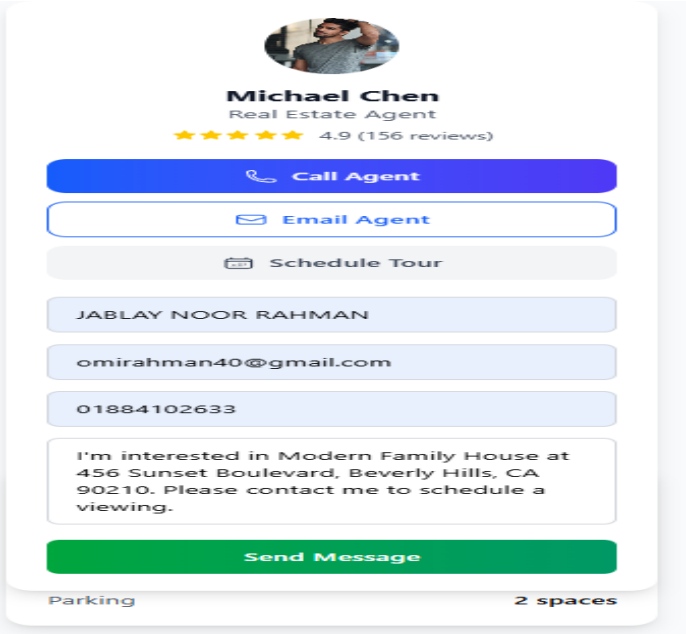


Figure 21: Agent Appointment Booking Page

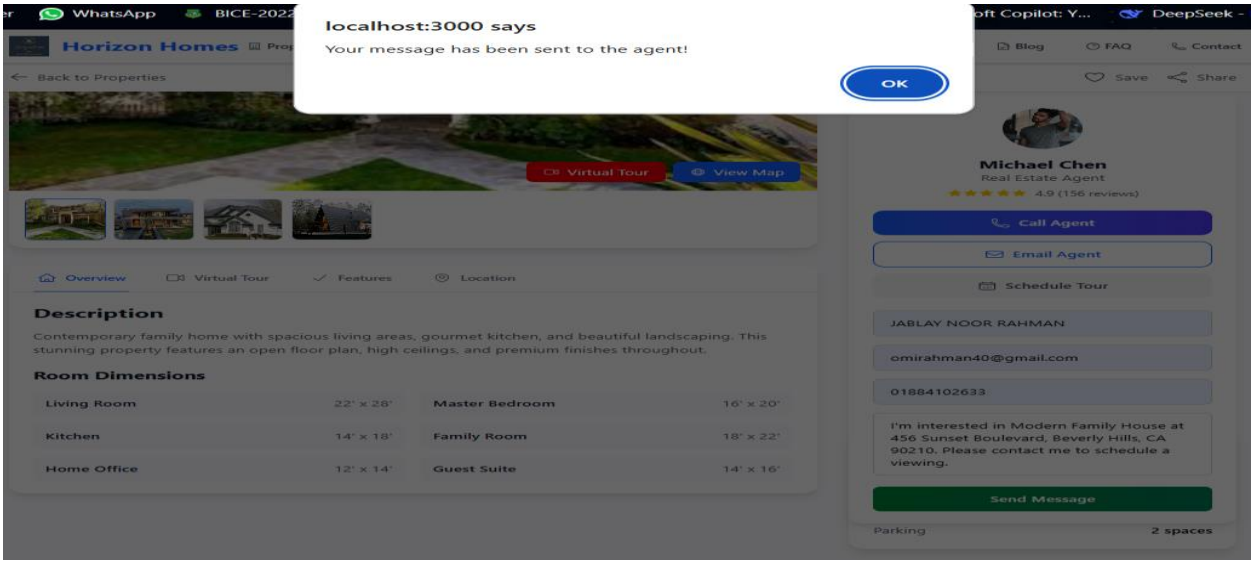


Figure 22: Agent Appointment Booking Successfully

NEIGHBORHOODS:

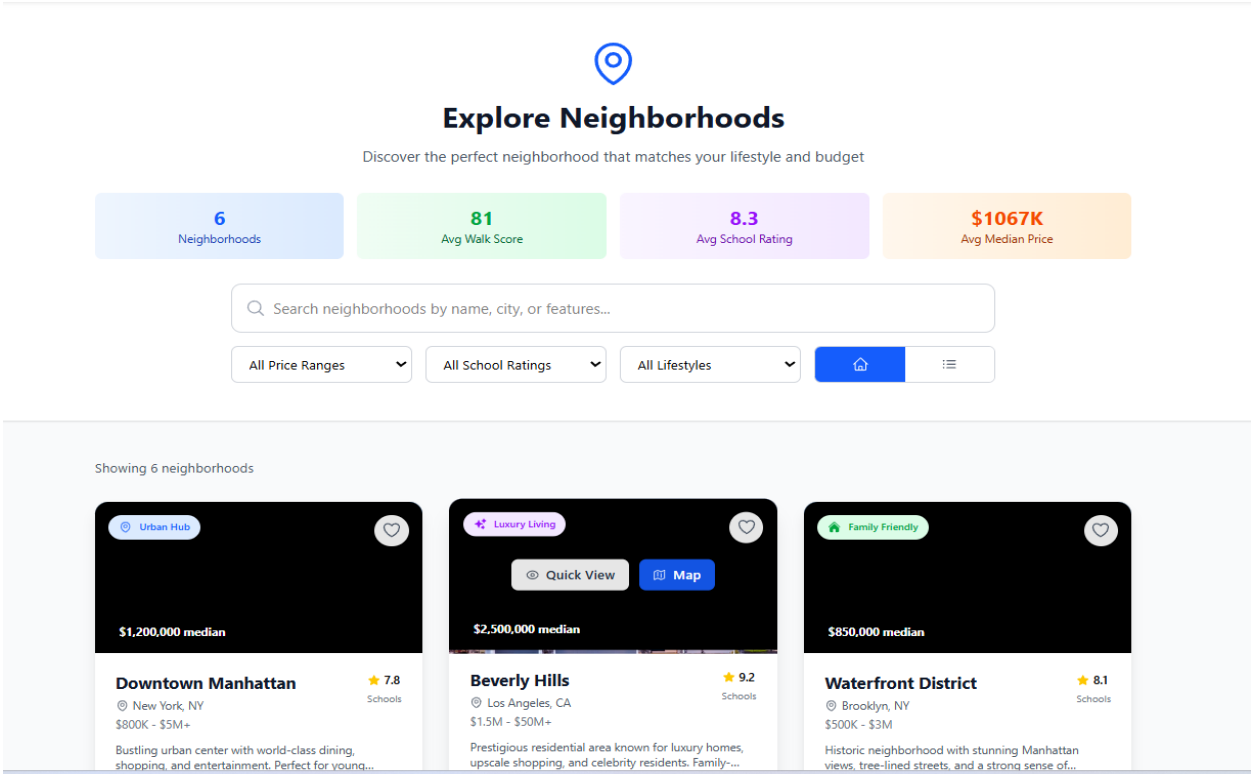


Figure 23: Neighborhoods Page

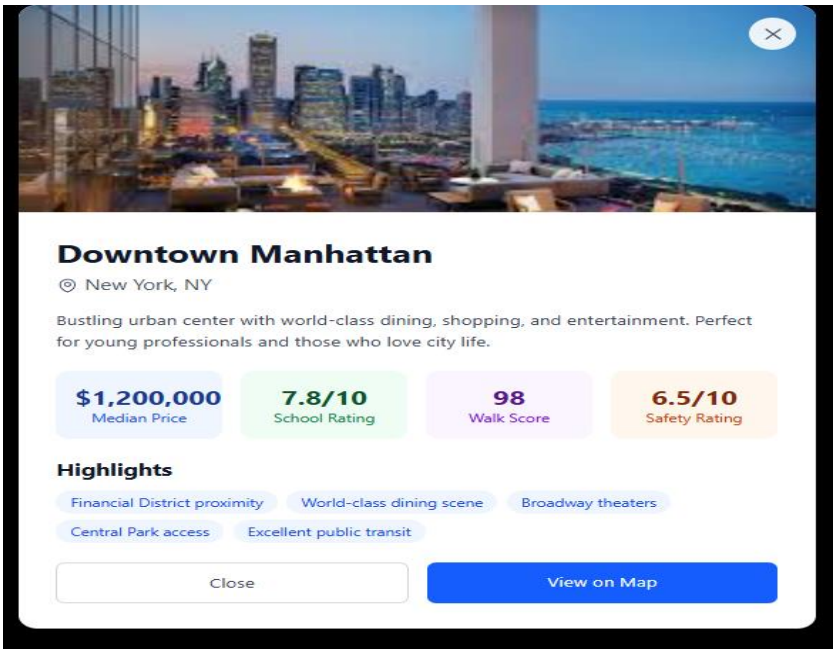


Figure 24: Neighborhoods Page (Quick View Details)

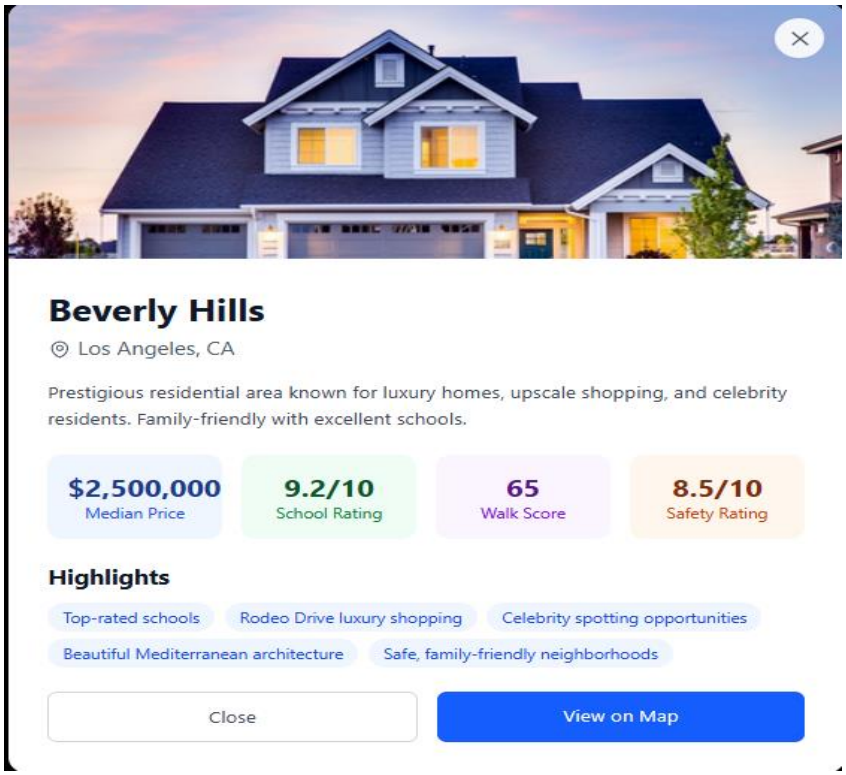


Figure 25: Neighborhoods Page (Quick View Details)

PROPERTIES COMPARE:

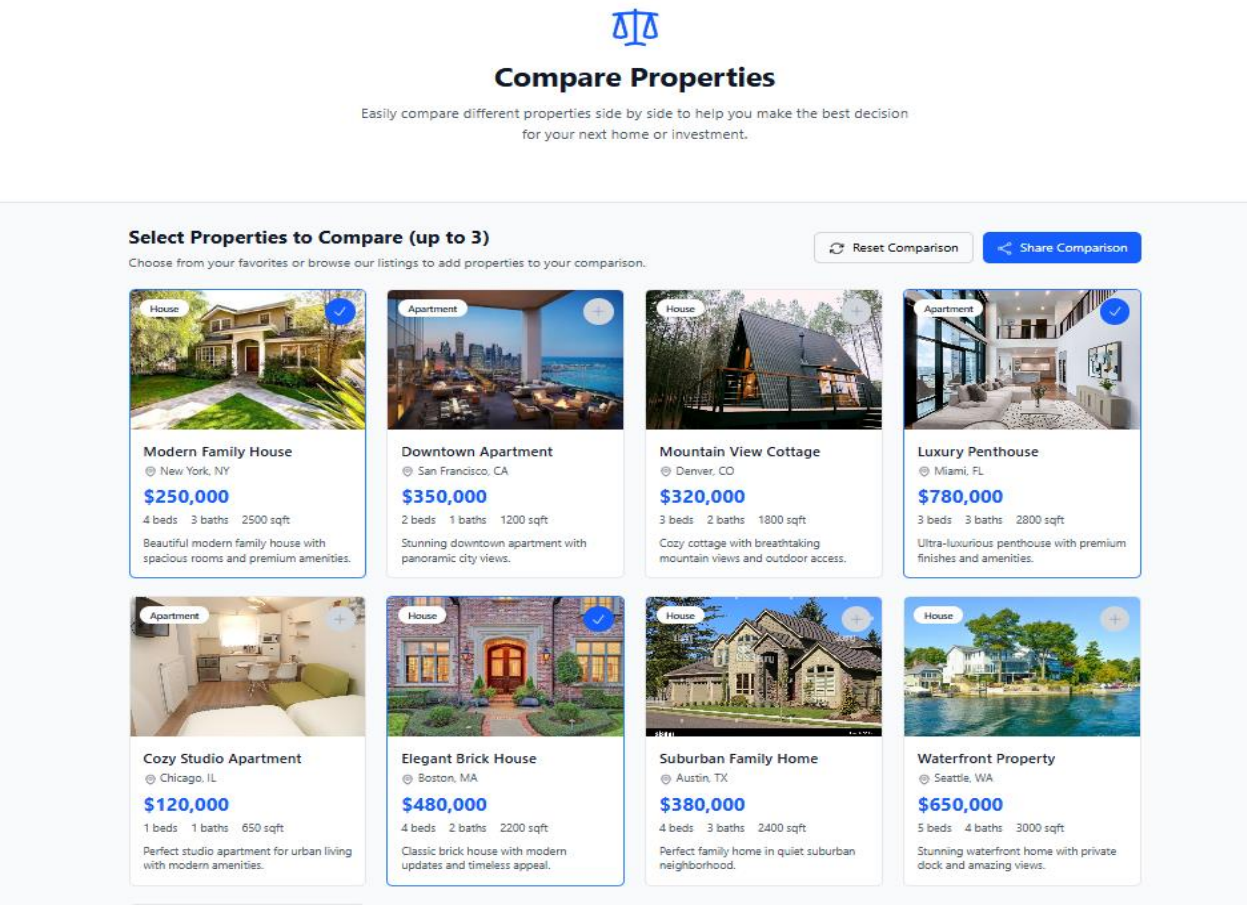


Figure 26: Properties Compare Page

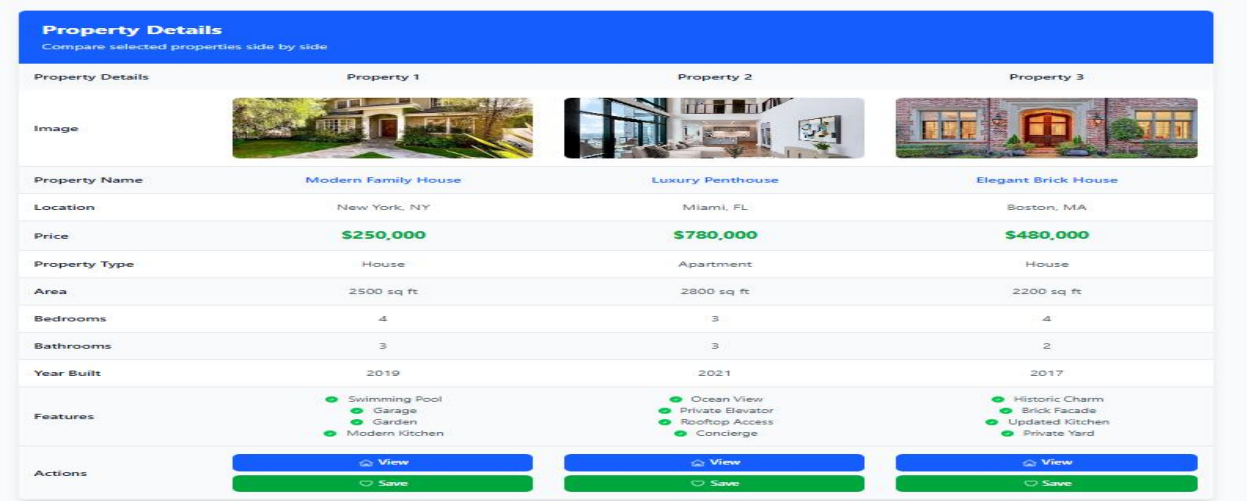


Figure 27: Slected Properties Comparing

MORTAGE CALCULATOR:

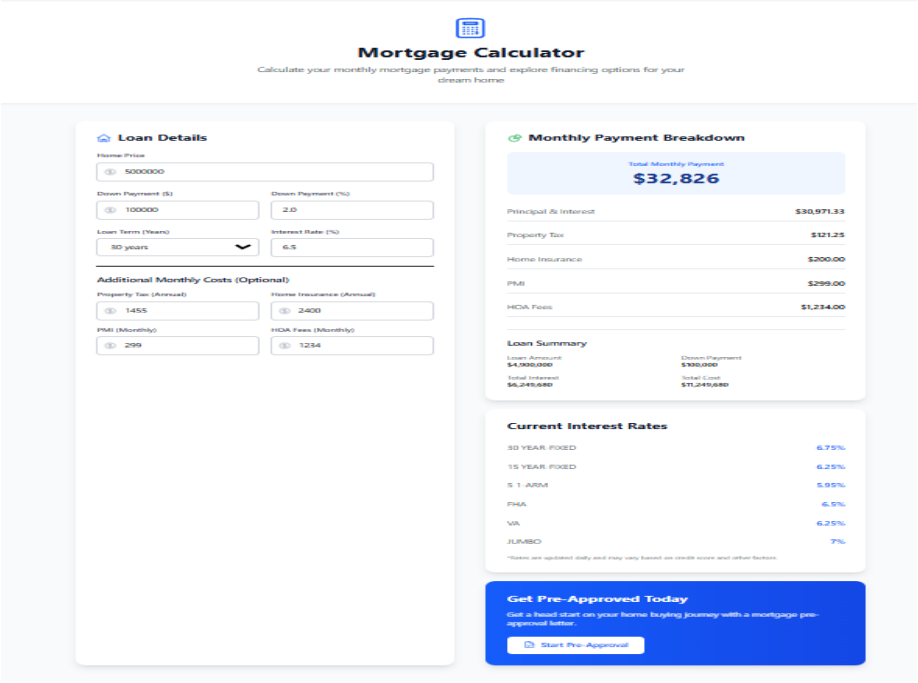


Figure 28: Mortage Calculator Page

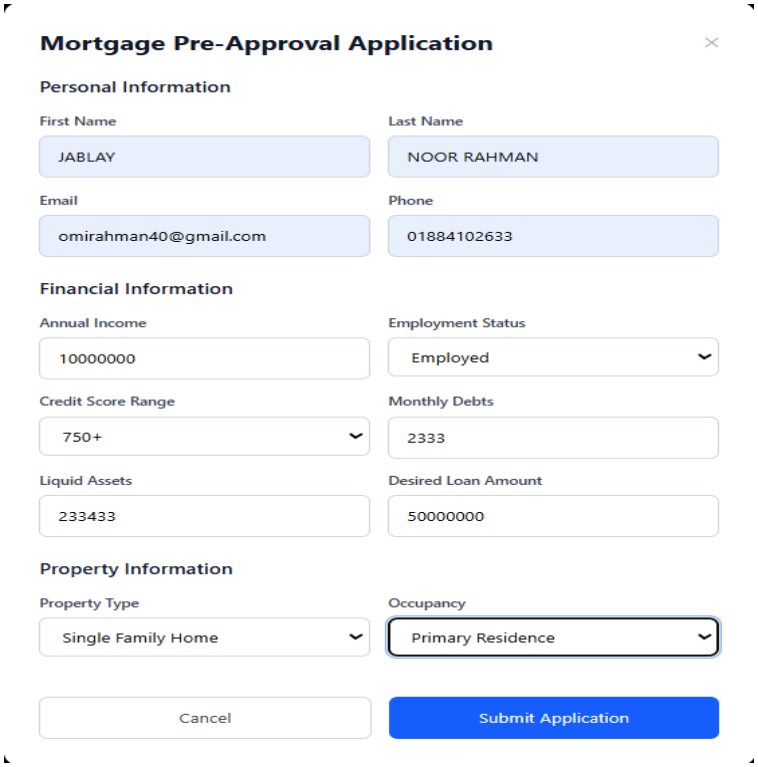


Figure 29: Mortage Pre-Approval Form Submission

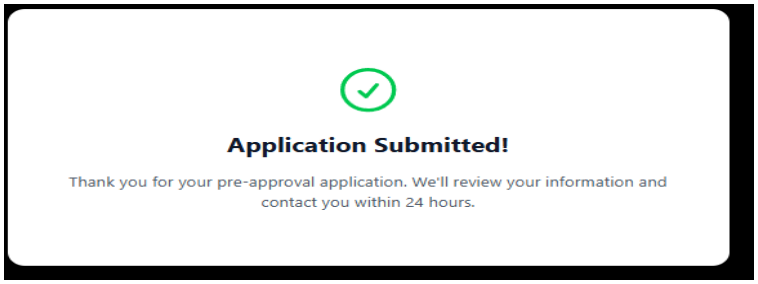


Figure 30: Mortage Pre-Approval Form Sumitted Successfully

CLIENT REVIEWS & FEEDBACK:

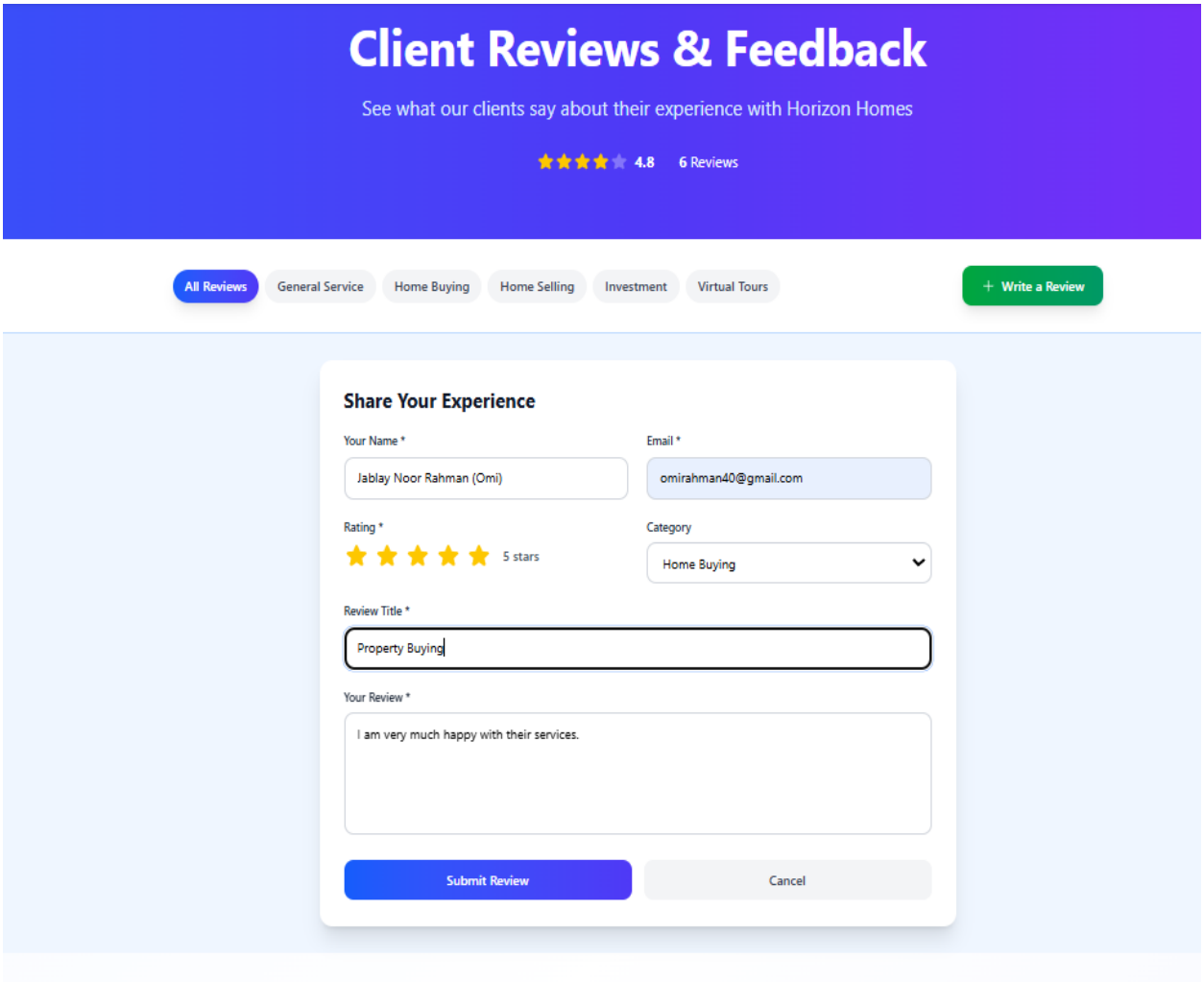


Figure 31: Client Review & Feedback Page

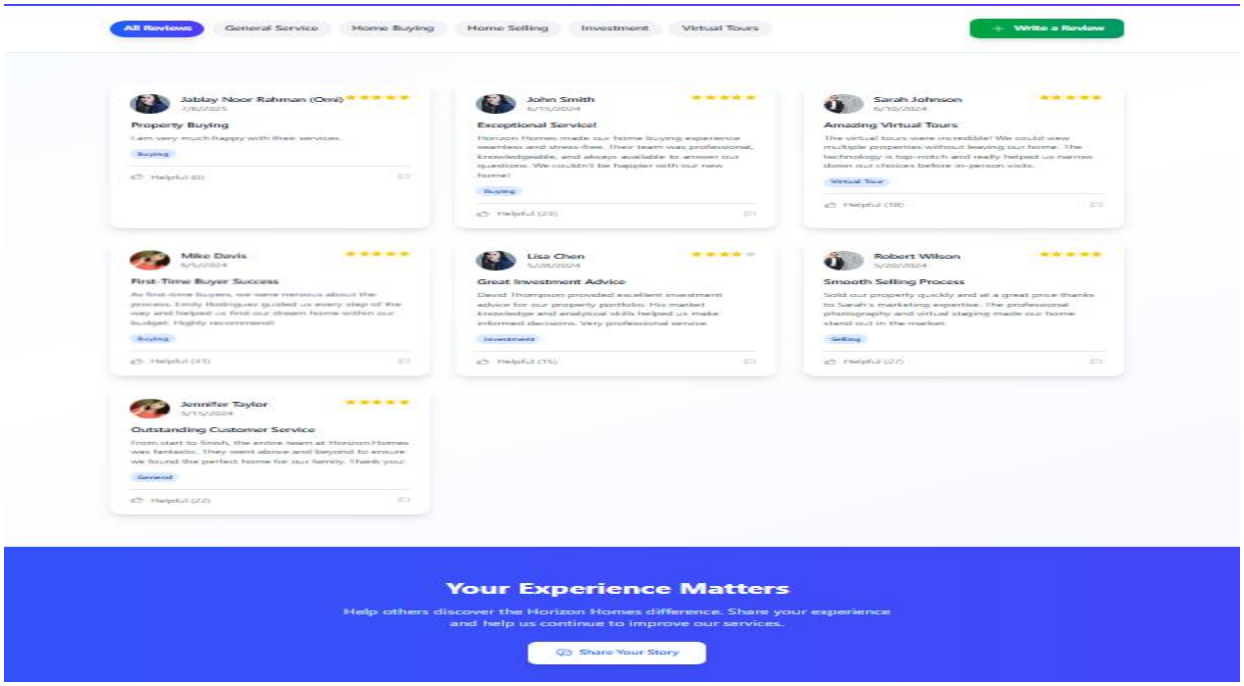


Figure 32: Client Review & Feedback Page

BLOG:

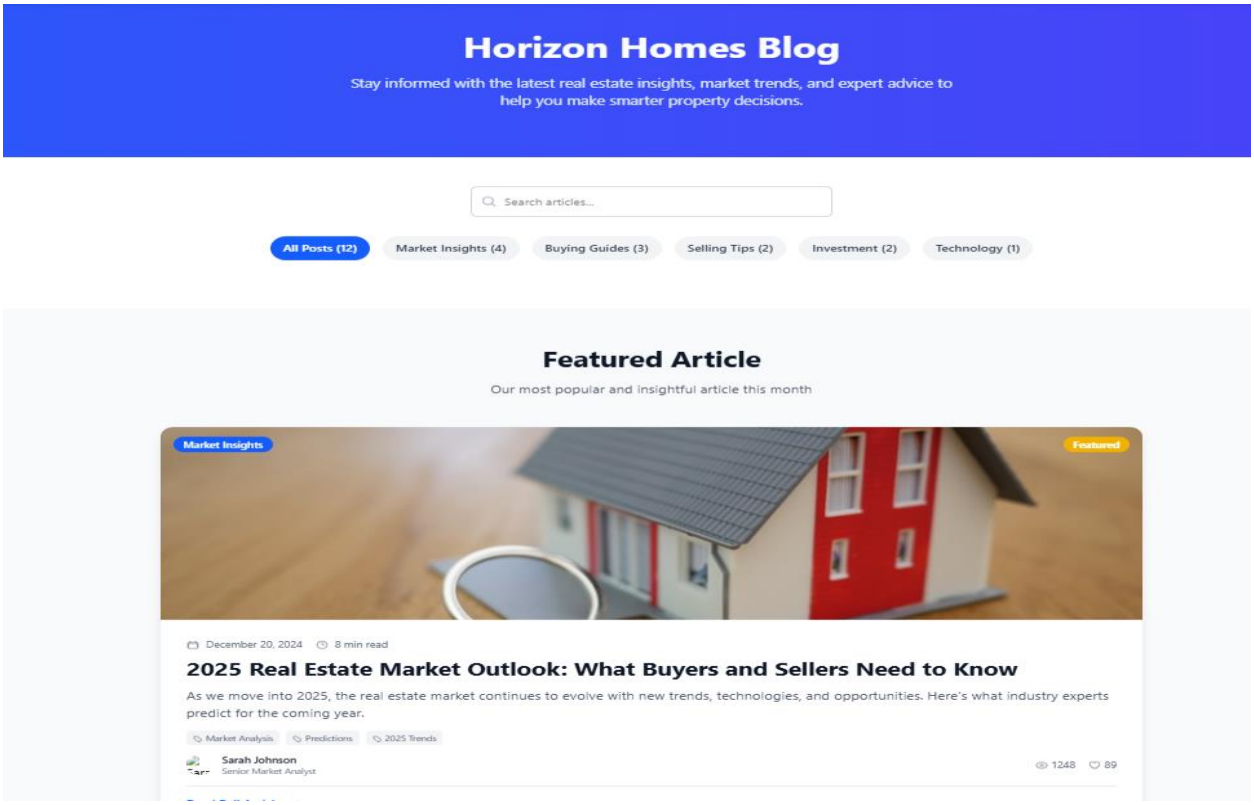


Figure 33: Horihon Homes Blog Page

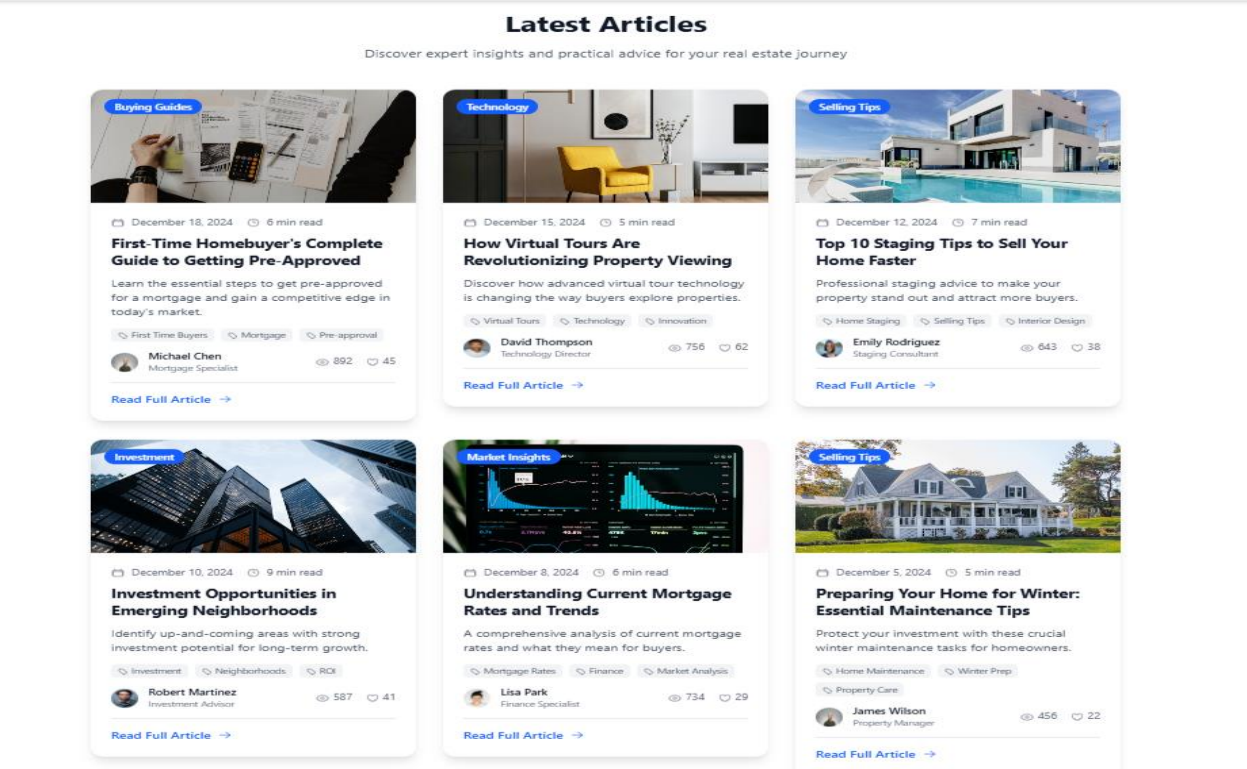


Figure 34: Latest Article On Blog Page

FAQ SECTION:

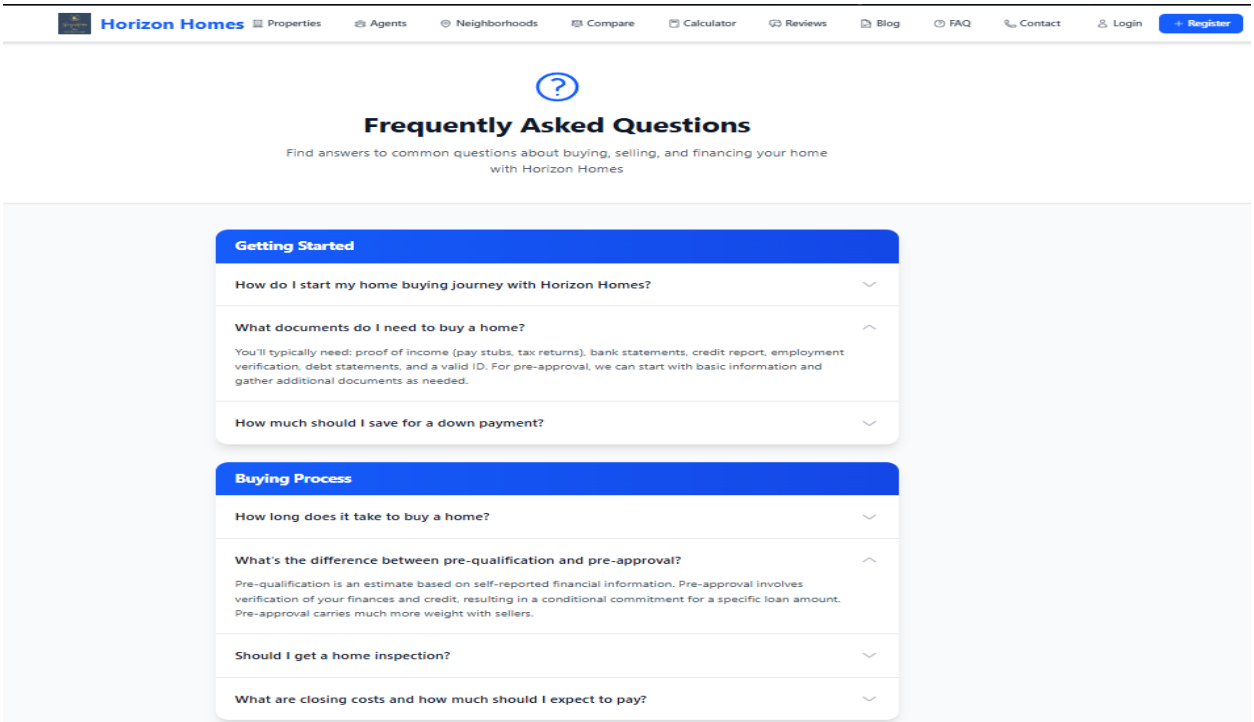


Figure 35: FAQ Page

CONTACT US:

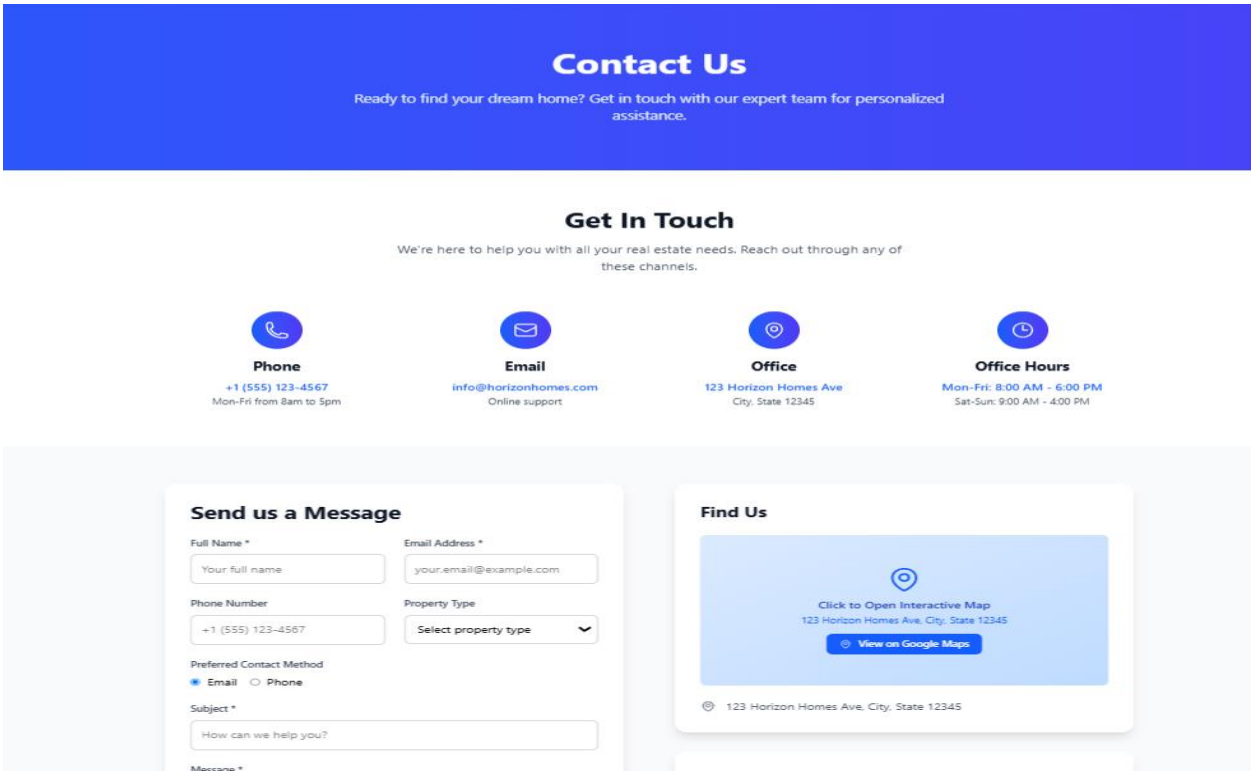


Figure 36: Contact Page

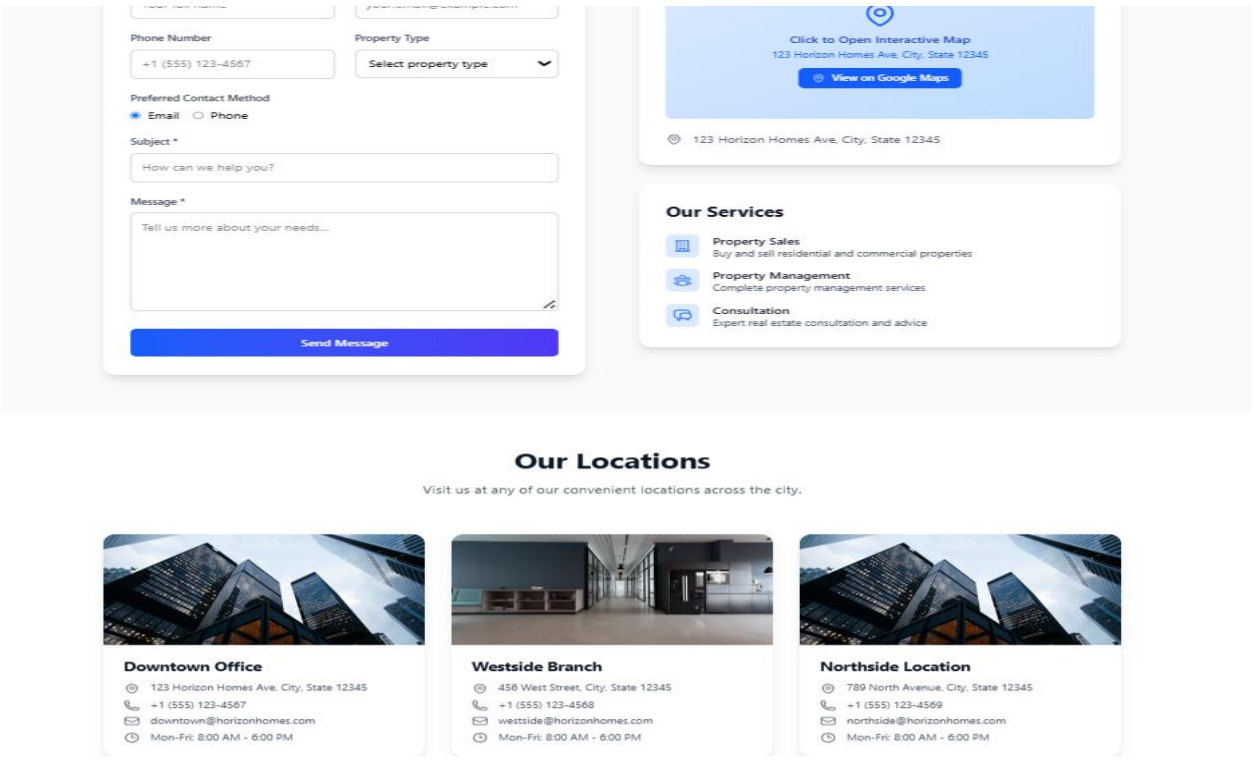


Figure 37: Contact Page

Databases:

- User Database:

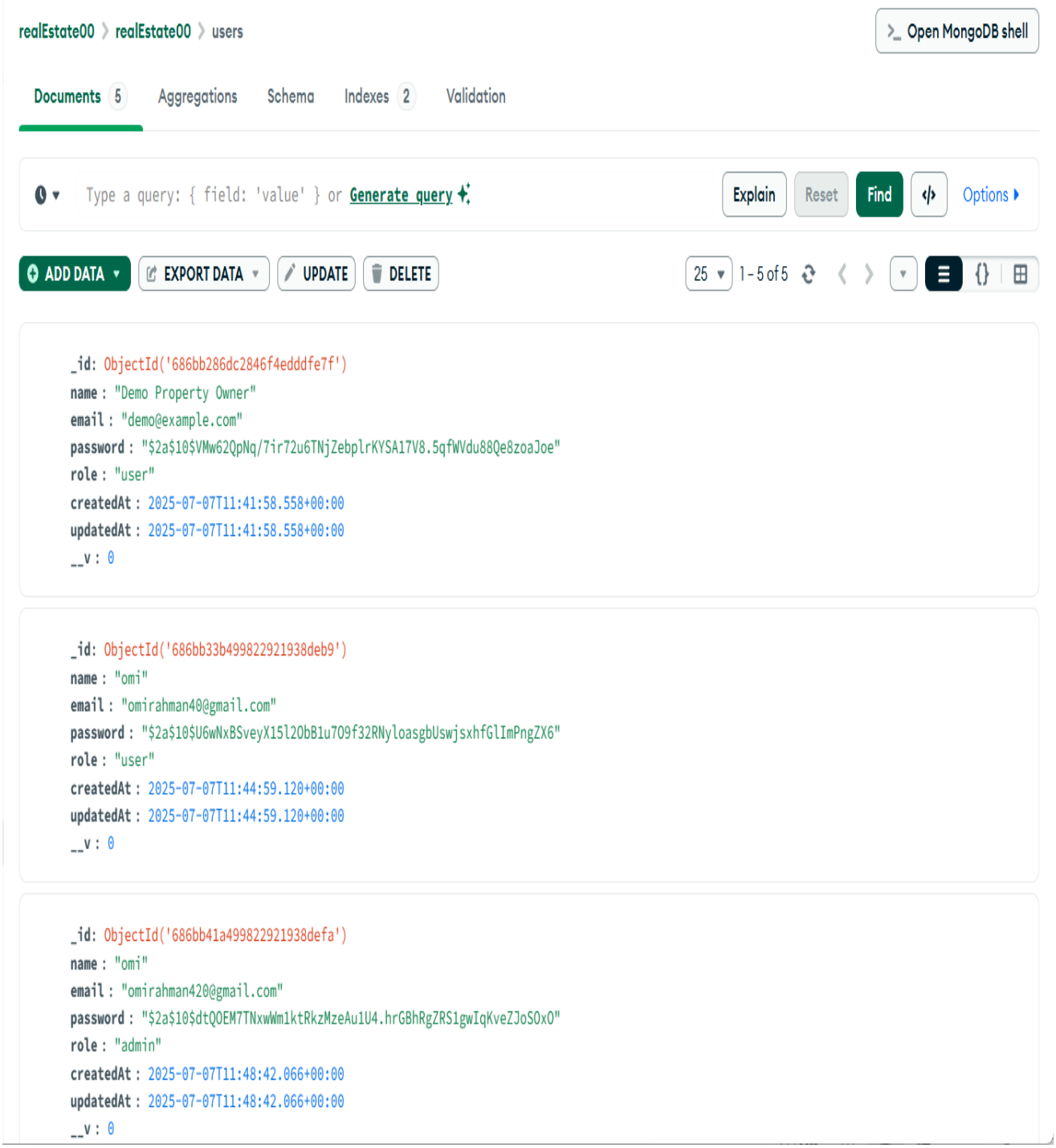


Figure 38: User Database

- Properties Database:

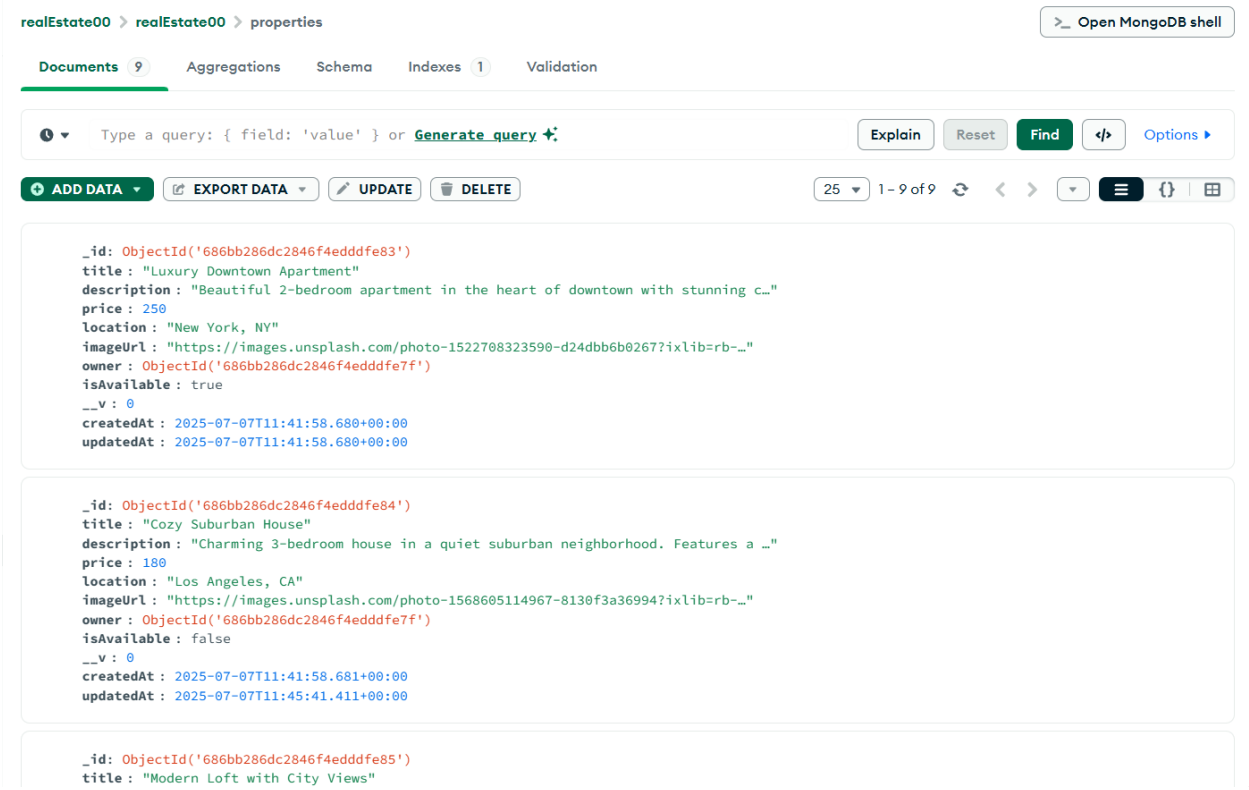


Figure 39: Properties Database

- Bookings Database:

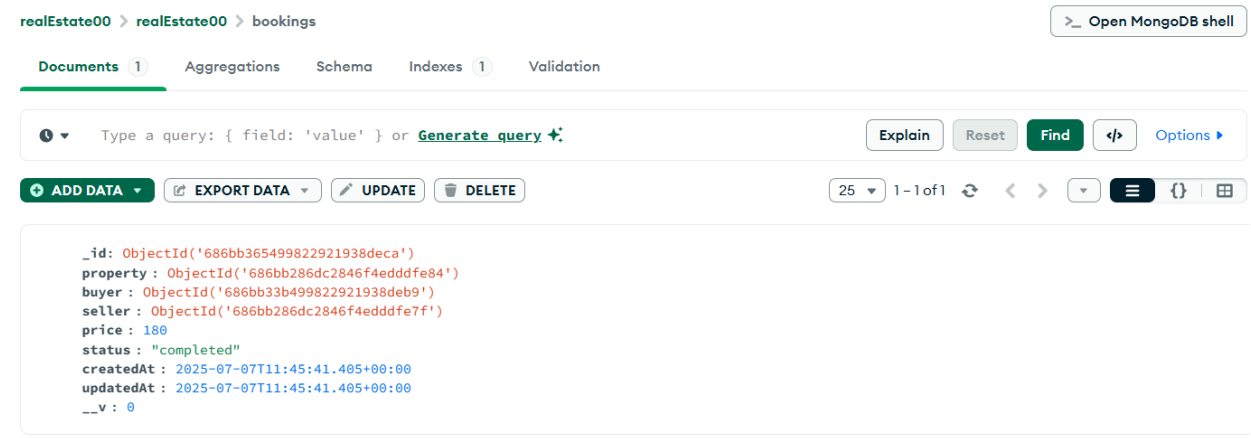


Figure 40: Booking Database

8. Challenges Faced and How We Solved Them:

While developing Horizon Homes, we encountered several key challenges typical of building a real estate platform, including issues related to real-time data management, user roles, property data management, and performance optimization. Below are the core challenges and how we solved them:

- Real-Time Data Synchronization Between Frontend and Backend

Challenge:
In a real estate platform like Horizon Homes, it's crucial that property listings, booking statuses, and availability are always up-to-date. Ensuring real-time synchronization between the frontend and backend was a major challenge. If data is outdated or inconsistent, users might book a property that's already taken, or see incorrect information.

Solution:
We utilized MongoDB Change Streams to monitor changes in the database. This allowed us to implement real-time updates on the frontend without needing page refreshes. When a new property was added, a booking was confirmed, or a property was updated, the changes were reflected instantly across all connected clients. This was critical for ensuring that property data was always current.

We also employed AJAX requests and React's state management to periodically fetch new data and keep the user interface synchronized with the backend.

- User Authentication and Role-Based Access Control

Challenge:
Horizon Homes needed secure user authentication and role-based access to ensure that different users (visitors, agents, and admins) had access to only the features they were authorized to use. Managing access and ensuring users were properly authenticated and authorized to perform specific actions was critical for both functionality and security.

Solution:
We implemented JWT (JSON Web Tokens) for secure authentication. When a user logged in, the system generated a JWT token that was sent to the frontend. This token was used to authenticate subsequent requests to the backend, ensuring that users could only perform actions they were authorized for.

For role-based access control, we used middleware on the backend to verify the user's role (Visitor, Agent, Admin) based on the JWT token before allowing access to specific routes. For instance:

- Visitors could only view property listings and use the mortgage calculator.
- Agents could list and manage properties, view their bookings, and respond to users.
- Admins had full access to manage all users, approve properties, and oversee bookings.

This approach ensured that only the appropriate users could interact with specific parts of the system.

- Property Management and Data Integrity

Challenge:
Managing property data effectively and ensuring its integrity was another significant challenge. Property listings in a real estate system involve varied types of data (text, images, prices, availability). Moreover, ensuring that all property data was correctly updated and consistent across the platform required careful planning.

Solution:
We used MongoDB as the database, which provided flexibility to handle complex and dynamic property data. For example, properties can have different types of features and fields (e.g., houses vs apartments), so a flexible schema was required.

To ensure data integrity:

- We used Mongoose to define and enforce schemas for each type of data (e.g., Property, User, Booking) to validate the data before saving it to the database.
- For CRUD operations (Create, Read, Update, Delete), we provided full administrative control through the backend routes for agents and admins to manage property listings.
- Admins had the ability to review and approve listings submitted by agents before they were made public, ensuring that all property data was correct and validated before appearing on the platform.

This solution allowed us to efficiently manage and maintain the integrity of property data.

- Search and Filtering Efficiency

Challenge:
Real estate platforms often contain large amounts of property data. Ensuring that the search and filtering systems were efficient and responsive, especially when dealing with complex property criteria (e.g., price, location, amenities), was a critical challenge.

Solution:
We used MongoDB's powerful querying system to create efficient search filters based on various property attributes like price, location, and property type. This ensured that property search results were filtered and sorted quickly, even with a large volume of listings.

- Indexes were created on commonly searched fields such as price, location, and availability, which improved search query performance significantly.
- The frontend used React's state management to update the property listings dynamically as users applied filters, making the experience seamless.

This approach helped us deliver a fast and efficient search system, even as the platform scaled.

- Handling Property Images and File Uploads

Challenge:
Managing large images for each property, while ensuring fast load times and a smooth user experience, was another challenge. Property images are often large files, and it's essential that the application can handle uploads efficiently without affecting performance.

Solution:
To handle image uploads and optimize performance, we used a cloud storage solution (such as AWS S3 or a similar service). This allowed us to store images externally, ensuring they could be served efficiently without burdening the application's primary server.

Additionally:

- Image compression techniques were applied before uploading to reduce file size without compromising quality.
- Lazy loading was implemented for images on the frontend, meaning images would only load when they were about to appear on the user's screen, improving the initial load time of the page.

By leveraging cloud storage and optimizing images, we improved performance and ensured a fast, user-friendly experience.

- Data Validation and Error Handling

Challenge:
Proper data validation was crucial to prevent users from entering incorrect or incomplete information when adding new properties, booking properties, or registering. Similarly, ensuring that the application gracefully handled errors and provided meaningful feedback to users was a challenge.

Solution:
We used Mongoose for schema validation, which allowed us to define strict validation rules for property data, such as required fields, data types, and custom validation (e.g., price must be a positive number). This prevented invalid data from being saved to the database. On the front end, we implemented form validation to catch errors before submitting data to the backend. For example, users were prompted to enter all necessary details when submitting a property listing or making a booking. For error handling, we implemented try-catch blocks and middleware on the backend to ensure that any unexpected issues were handled gracefully. Errors were logged, and user-friendly messages were provided on the front-end to inform users of any issues, such as missing fields or failed bookings.

The Horizon Homes project presented several unique challenges, but by leveraging the appropriate technologies (MongoDB, React, JWT, etc.) and carefully planning each aspect of the development process, we were able to overcome these challenges. By focusing on real-time data synchronization, secure authentication, efficient property management, and a responsive, mobile-

first design, we were able to deliver a seamless and user-friendly platform for managing real estate listings and bookings.

9. Conclusion:

The development of Horizon Homes, a real estate web application, significantly contributes to the real estate industry by simplifying the property search, listing, and booking processes, while enhancing the overall user experience. By successfully integrating key features such as secure user authentication, advanced search functionalities, real-time property management, and role-based access, Horizon Homes improves the efficiency of both property seekers and property owners. It facilitates seamless property reservations and provides a comprehensive platform for users to manage and interact with property listings.

Throughout the creation of Horizon Homes, we learned hands-on experience in full-stack web development, focusing on both front-end and back-end technologies. We applied our technical skills in HTML, CSS, JavaScript, React, and backend frameworks like Node.js and Express to create a dynamic, interactive, and secure platform. The project allowed us to explore real-time data management, implementing technologies like MongoDB to handle property listings, bookings, and user data. By designing and building a responsive, mobile-first user interface, we gained insights into modern web design and user experience practices, ensuring accessibility across devices.

Additionally, we learned the critical importance of data security and privacy in the context of a real estate platform. With the implementation of secure JWT (JSON Web Token) authentication and bcrypt password hashing, Horizon Homes ensures user data is protected from unauthorized access. Furthermore, performance optimization, through techniques like lazy loading, code splitting, and real-time data synchronization, enhances the responsiveness and speed of the application, offering a seamless experience for users, even when browsing large sets of property data.

The Horizon Homes project also provided valuable insights into the real estate industry itself, particularly around the challenges faced by property buyers, sellers, and agents. We gained a deeper understanding of the importance of accurate property data, transparent listings, and a simplified booking process. Through role-based access, we learned how to manage different user types—visitors, agents, and admins—while ensuring each user has access to the relevant features based on their role. The platform also highlighted the significance of effective communication and user trust in a digital real estate ecosystem.

In conclusion, the development of Horizon Homes is a valuable project that equipped us with practical skills in web development, database management, and system integration. It enhanced our understanding of both the technical aspects of building a real estate platform and the business side of the real estate industry. We not only improved our problem-solving and project management skills but also deepened our expertise in creating user-centric, secure, and high-performance applications. With Horizon Homes, users can easily find, book, and manage properties, while property owners and agents can effectively manage listings and bookings, ultimately improving the overall real estate experience for everyone involved.