**1. Introduction**

* **Project Overview:** This React application is a real estate website designed to showcase and facilitate the search and discovery of properties. The application allows users to browse properties, filter search results, read related blog posts, and contact the real estate company for inquiries. The properties data is fetched from Pixxi CRM.
* **Target Audience:** The primary target audience for this application includes potential buyers, sellers, and renters of real estate properties, specifically focusing on the Dubai market (implied from the code's use of Dubai-related content).
* **Key Features:**
  + Property Search: A robust search functionality allowing users to filter and find properties based on various criteria (e.g., property type, location, price range, number of bedrooms).
  + Property Listings: Detailed listings providing comprehensive information about available properties, including images, descriptions, and key features.
  + Blog Section: A section dedicated to real estate-related blog posts, providing valuable insights and information to users.
  + Contact Form: A contact form allowing users to easily reach out to the company for inquiries or assistance.
  + Responsive Design: Adapts to various screen sizes for optimal usability across desktops, tablets, and mobile devices.

**Project Structure:**

The project follows a standard React/Vite structure:

* **public:** Contains static assets like images, fonts, and SVG. This directory is served directly by Vite.
* **src:** Contains the source code. The further breakdown into Components, Context, Hooks, and Pages is organized maintainability.
* **Components:** Houses reusable UI components. There's a good separation of concerns here, with components grouped logically (e.g., Hero, Input Field, Loaders).
* **Context:** Manages application state using React Context API (Navbar Context, Property Data Context).
* **Pages:** Contains the application's different routes (e.g., About Us, Blogs, Properties). Each page likely contains multiple components.
* **Configuration Files:** package. json, vite.config.js, tailwind.config.js, postcss.config.js, eslint.config.js manage dependencies, build process, styling, and linting respectively.
* **Other Files:** env, gitignore, htaccess, handle environment variables and server configuration.

**Component Documentation**

This section will be the bulk of your document. For every component folder (e.g., Hero, Property, Search Bar), there is a separate subsection.

* File Name: (e.g., HeroPropertiesSection)
* Description: A concise explanation of the component's purpose.
* Functions/Method: A table listing the component's state variables, their data types, and how they are used.

File Name: Hero/HeroSectionModal.jsx

|  |  |
| --- | --- |
| Description | His component renders a modal-like hero section with a slideshow of images that adapt to different screen sizes. It includes a top navigation bar and image indicators. The image slideshow automatically cycles through images every 5 seconds. |
| Functions/Methods | * HeroPropertiesSectionModal({ children }): The main component that renders the hero section. It takes children as a prop, allowing other components to be rendered within the hero section. * useCurrentScreenSize(): A custom hook that detects the current screen size based on predefined breakpoints. * getImagesForScreen(): Returns an array of image filenames based on the current screen size. * getScreenSize(): Helper function determining screen size based on window width and breakpoints. * handleClick(index): Updates the currentImage state to display a specific image from the slideshow. * TopNavigationTab(): Renders a top navigation bar for smaller screens. * TopNavigationTabLarge(): Renders a top navigation bar for larger screens. |

File Name : Hero/HeroSectionModal2.jsx

|  |  |
| --- | --- |
| Description | Similar to HeroSectionModal.jsx, this component also renders a hero section with an image slideshow, but with a simpler design and a shorter slideshow interval (2 seconds). |
| Functions/Methods | * HeroPropertiesSectionModal2({ children }): The main component rendering the simplified hero section. Takes children as a prop. * useCurrentScreenSize(): (Same as in HeroSectionModal.jsx) A custom hook that detects the current screen size. * getImagesForScreen(): Returns an array of image filenames based on the current screen size. * handleClick(index): Updates the currentImage state. * TopNavigationTab(): (Same as in HeroSectionModal.jsx) Renders a top navigation bar for smaller screens. * TopNavigationTabLarge(): (Same as in HeroSectionModal.jsx) Renders a top navigation bar for larger screens. * getScreenSize(): (Same as in HeroSectionModal.jsx) Helper function determining screen size. |

File Name: HeroBlogSection/HeroSection.jsx

|  |  |
| --- | --- |
| Description | This component renders the hero section for the blog page. It displays a heading and uses an image as the background. It also includes a top navigation bar. |
| Function/Methods | * HeroBlogSection({ HeroText }): Renders the Blog hero section. * updateParentHeight(): Updates the height of the parent container based on the video/image height. * updateVideoSource(): Sets the video source based on screen size. * TopNavigationTab(): Renders a top navigation bar for smaller screens. * TopNavigationTabLarge(): Renders a top navigation bar for larger screens. |

Above is same for the : HeroCareerSection , HeroContactSection, HeroPropertiesSection ,HeroServiceSection

File Name: HeroPropertiesSection/SearchBar.jsx

|  |  |
| --- | --- |
| Description | A search bar component that allows users to filter properties based on various criteria and navigate to the search results page. It fetches a list of developers. |
| Function/Methods | * SearchBar(): The main search bar component. * handleFilterChange(filterType, value): Updates the selected filters. * fetchDevelopers(): Asynchronously fetches a list of developers from an API. * SearchProperties(): Navigates to the search results page with the selected filters. |

File Name: HeroSearchProperties/HeroSection.jsx

|  |  |
| --- | --- |
| Description | This component displays a list of properties fetched from an API based on search criteria. It uses pagination for better performance. |
| Function/Methods | * ): The main component rendering the search results. * fetchProperties(): Fetches properties from the API, handling loading and pagination. * handleClickItem(propertyId, Type, DeveloperLogo): Navigates to a property detail page. * handlePageClick({ selected }): Handles pagination clicks. * PropertyListingNEW({ properties, handleClickItem, Type }): Renders a list of new properties. * PropertyListingRENT({ properties, handleClickItem, Type }): Renders a list of buy/rent properties. * formatPrice(price): Formats the price to display K or M for thousands and millions. |

File Name: ScrollProperty.jsx

|  |  |
| --- | --- |
| Description | This component fetches and displays a horizontally scrollable list of properties. It handles different property types ("New Projects", "Buy", "Rent"), pagination, loading states, and drag-to-scroll functionality. The component also conditionally renders navigation arrows and a dashed line depending on the Image prop. It uses a custom NavBar component for property type selection. |
| Function/Methods | * ScrollProperty({ Page, Image = { default: false } }): The main component. Page specifies the page number for pagination. Image is a prop object that controls conditional rendering (e.g., for hero sections). It fetches property data, conditionally renders a loader, and renders either PropertyListingNEW or PropertyListingRENT based on the selected property type. * fetchProperties(): Fetches property data from a specified API endpoint using fetch. The API call includes parameters for listing type, page number, and page size. * handleClickItem(propertyId, Type, DeveloperLogo): Navigates to a property detail page using react-router-dom. * PropertyListingNEW({ properties, handleClickItem, Type, User, setUser, Image }): Displays a horizontally scrollable list of new properties. Implements drag-to-scroll functionality using mouse events and provides next/previous buttons. It also renders the NavBar component. * handleMouseDown(e): Starts drag-to-scroll functionality. * handleMouseLeave(), handleMouseUp(): Ends drag-to-scroll functionality. * handleMouseMove(e): Updates scroll position based on mouse movement during dragging. * formatPrice(price): Formats a price value to display in K (thousands) or M (millions). * handleNext(): Scrolls to the next property in the list. * handleBack(): Scrolls to the previous property in the list. * PropertyListingRENT({ properties, handleClickItem, Type, User, setUser, Image }): Similar to PropertyListingNEW, but displays rent/buy properties and handles different property types. * NavBar({ User, setUser, Image }): A custom component rendering navigation buttons for selecting property types ("Buy", "New Projects", "Rent"). Uses motion.div for animation. * WhiteBackground({ user, setUser, children }): A component for styling the navigation buttons. Uses motion.div for hover animation. |

File Name: SearchBar/SearchBar.jsx

|  |  |
| --- | --- |
| Description | This component renders a search bar with filtering capabilities. It fetches a list of developers and allows users to select developers, property types, and the number of bedrooms to filter search results. The component updates the URL query parameters to reflect the selected filters without causing navigation and updates the PropertyData in context. |
| Function/Methods | * SearchBar(): The main component. It manages state for filters, fetches developer data, and handles filter selection. It renders a search input (SearchInput) and filter options (CustomDropFilter). * useEffect(): Fetches the list of developers on component mount. Uses a second useEffect to update UseFilter state based on the PropertyData from context. There are additional useEffect hooks to update the QuerySelect states (QueryDeveloper, QueryPropertyType, QueryBedRoomNum) based on changes in PropertyData from context. * handleFilterChange(filterType, value): Updates the selected filters state. * SearchProperties(): Updates the URL with new query parameters based on the selected filters. Uses window.history.pushState to avoid a full navigation. Updates PropertyData in context. * CustomDropFilter({ Name, Value, developers, handleFilterChange, QuerySelect }): A reusable component for creating dropdown filters. It manages the dropdown's open/close state, handles filtering based on a search term, and updates the selected filters. * handleSelectChange(value): Handles the selection of items in the dropdown filter. * handleRemoveDeveloper(value): Handles the removal of items from the dropdown filter. * useEffect(): Manages the dropdown's visibility based on clicks inside and outside of the dropdown. |

**Context Documentation**

This section will be the bulk of your document. For every component folder (e.g., Hero, Property, Search Bar), there is a separate subsection.

* File Name: (e.g., NavBarContext.jsx)
* Description: A concise explanation of the Context's purpose.
* Functions/Method: A table listing the component's state variables, their data types, and how they are used.

File Name: NavBarContext.jsx

|  |  |
| --- | --- |
| Description | This file defines a React context for managing the currently selected navigation item (property type: "New Projects", "Buy", or "Rent"). It uses useSearchParams from react-router-dom to synchronize the context with URL parameters. |
| Function/Methods | * NavBarProvider({ children }): This component provides the context value. It uses the useSearchParams hook to read URL parameters. If listingType exists in the URL, the User state is updated accordingly; otherwise, a default value is used. * useNavBar(): A custom hook that allows components to access and update the User state (selected navigation item) from the context. |

File Name: PropertyDataContext.jsx

|  |  |
| --- | --- |
| Description | This file defines a React context for managing property search data. It uses useSearchParams to read URL parameters and updates its state accordingly. It converts query parameters into appropriate data structures (arrays or single values). |
| Function/Methods | * PropertyDataProvider({ children }): This component provides the context value. It reads URL parameters using useSearchParams. It processes these parameters, converting comma-separated strings into arrays and ensuring that the listingType always has a value (defaulting to "NEW"). The processed data is then stored in the PropertyData state. * usePropertyData(): A custom hook that allows components to access and update the PropertyData state from the context. This provides a way for components to share and manage the property search criteria. |

**Page Documentation :**

This section will be the bulk of your document. For major page folder (e.g., Properties), there is a separate section.

* File Name: (e.g., Properties.jsx)
* Description: A concise explanation of the page purpose.
* Functions/Method: A table listing the component's state variables, their data types, and how they are used.

Properties

File Name: Blog.jsx

|  |  |
| --- | --- |
| Description | This component renders a section displaying blog posts. It features a navigation bar (NavBar) for selecting blog categories ("Guides" or "Blog") and a scrollable list of blog entries (BlogSection). |
| Function/Methods | * Blog(): The main component, which renders the blog section header, navigation, and the BlogSection. * NavBar({ User, setUser }): A navigation component to switch between blog categories. It uses a WhiteBackground component to style the selected category. * WhiteBackground({ user, setUser, children }): Styles the navigation buttons using motion.div for hover effects. * BlogSection(): Renders a horizontally scrollable list of blog posts, including images, titles, and descriptions. It implements drag-to-scroll and provides "Read More/Show Less" functionality for descriptions. * toggleDescription(id): Toggles the expanded/collapsed state of a blog post's description. * handleMouseDown(e), handleMouseLeaveOrUp(), handleMouseMove(e): Handle drag-to-scroll functionality. * truncateCombinedText(title, description, limit = 80): Truncates the combined title and description to a specified character limit. |

File Name: ExploreProperties.jsx

|  |  |
| --- | --- |
| Description | This page displays a section showcasing featured properties, allowing users to explore them. It includes a navigation bar (NavBar) for selecting property types and utilizes the ScrollProperty component to render a scrollable list of properties. |
| Function/Methods | * ExploreProperties(): Renders the "Explore Properties" section, including a header, navigation bar, and property listings via ScrollProperty. * NavBar({ User, setUser }): A navigation component for selecting property types ("Buy", "New Projects", "Rent"). It uses the WhiteBackground component for styling. * WhiteBackground({ user, setUser, children }): Styles the navigation buttons using motion.div for hover effects. |

File Name: MeetTeam.jsx

|  |  |
| --- | --- |
| Description | This component displays information about the team, including advisors' names, titles, and contact information. It uses a sidebar for filtering advisors by location and a main area to display advisor information. |
| Function/Methods | * MeetTeam(): The main component, rendering the team header, navigation arrows, sidebar (Sidebar), and advisors' details (Advisors). * Sidebar(): Renders a sidebar with a list of locations to filter advisors. * handleClick(place): Updates the selected location in state. * Advisors(): Displays cards with advisors' information (name, title, image, and contact). |

File Name: NavBar.jsx

|  |  |
| --- | --- |
| Description | This component renders a navigation bar with menu items. It includes a responsive design, adapting to different screen sizes with a toggle menu. |
| Function/Methods | * Navbar(): The main navigation bar component. * toggleMenu(): Toggles the visibility of the responsive menu. |

File Name: Podcasts.jsx

|  |  |
| --- | --- |
| Description | This component displays a list of podcasts or service offerings. Each podcast is presented as an accordion, revealing more details upon expansion. The component adapts its behavior based on the screen size, opening all accordions on larger screens and only allowing one open at a time on smaller screens. |
| Function/Methods | * Podcasts(): Renders the main podcast section header, navigation, and service boxes (ServiceBox). * handleAccordionClick(index): Handles the click on an accordion to toggle its state. * useEffect(): Adjusts accordion states based on screen size. * ServiceBox({ isOpen, onClick, service }): Renders a single service box with an accordion-like behavior. * BlogSection(): (Duplicate from Blog.jsx) Renders a horizontally scrollable list of blog posts. * toggleDescription(id), handleMouseDown(e), handleMouseLeaveOrUp(), handleMouseMove(e), truncateCombinedText(title, description, limit = 80): (Duplicate from Blog.jsx) Handles drag-to-scroll and description expansion/truncation. |

File Name: SearchBar.jsx

|  |  |
| --- | --- |
| Description | A search bar component offering filtering options for properties. It provides a basic search input and a filter toggle. The filtering logic is not fully implemented. |
| Function/Methods | * SearchBar(): The main component. * handleFilterChange(filterType, value): Placeholder for filter change handling (not implemented). * DropFilter({ Name, developers, handleFilterChange }): A reusable dropdown filter component. It handles selection and removal of filter options. It uses motion.div for animations. |

File Name: SearchProperties.jsx

|  |  |
| --- | --- |
| Description | This component renders a search page for properties. It uses context providers (NavBarProvider, PropertyDataProvider) and includes a hero section, navigation bar, search bar, and search results (HeroSearchSection). |
| Function/Methods | * SearchProperties(): The main component rendering the page. * useEffect(): Updates the PropertyUser state based on the listingType query parameter. * HeroText(): Renders the header text for the search page. * NavBar(): Renders a navigation bar for selecting property types. Uses WhiteBackground component for styling. * WhiteBackground({ user, setUser, children }): Styles the navigation buttons. |

File Name:

|  |  |
| --- | --- |
| Description |  |
| Function/Methods |  |

**III. API Interactions**

This section should detail the API endpoints used in the application (e.g., fetching properties, developers).

API Request 1: ScrollProperty (POST

* ScrollProperty({ Page, Image = { default: false } }): This is the main function of the component. It takes two props: Page for pagination and Image for conditional styling. It fetches property data using fetchProperties(), handles loading states, and renders either PropertyListingNEW or PropertyListingRENT based on the selected User property type.
* fetchProperties(): This asynchronous function fetches property data from the specified API endpoint using a POST request. It includes parameters for listingType, size (number of properties per page), and page number.
  + **Request Method:** POST
  + **Request URL:** https://dataapi.pixxicrm.ae/pixxiapi/v1/properties/Eplog Properties
  + **Request Body:**
  + {
  + "listingType": User[1],
  + "size": 10,
  + "page": Page
  + }
  + **Headers:**
  + Content-Type: application/json
  + X-PIXXI-TOKEN: FjuseQnHvSZy4jTqs8EN6uHfRz85YGv- //API TOKEN - REPLACE WITH PLACEHOLDER IN PRODUCTION
  + **API Token:** FjuseQnHvSZy4jTqs8EN6uHfRz85YGv- (REPLACE WITH PLACEHOLDER IN PRODUCTION)
  + **Expected Response:** A JSON object with a data property containing an array of property objects (DevelopersData.data.list). The statusCode property likely

**Endpoint 1: Fetch Developers**

* + **Method:** POST
  + **URL:** https://dataapi.pixxicrm.ae/pixxiapi/v1/developer/list
  + **Request Body (JSON):**
  + {
  + "page": 1,
  + "size": 10000
  + }
  + content\_copyUse code [with caution](https://support.google.com/legal/answer/13505487).Json
  + **Headers:**
  + Content-Type: application/json
  + X-PIXXI-TOKEN: FjuseQnHvSZy4jTqs8EN6uHfRz85YGv- (**This token must be replaced with a secure method of access in production.**)
  + **Response (Expected JSON):** A JSON object containing a list of developers. The statusCode indicates success (200) or failure. Example:
  + {
  + "statusCode": 200,
  + "data": {
  + "list": [
  + { "id": 1, "name": "Developer A" },
  + { "id": 2, "name": "Developer B" },
  + // ...more developers
  + ]
  + }
  + }
  + content\_copyUse code [with caution](https://support.google.com/legal/answer/13505487).Json
  + **Endpoint 2: Search Locations**
  + **Method:** POST
  + **URL:** https://dataapi.pixxicrm.ae/pixxiapi/v1/search/{query} (where {query} is the search term)
  + **Request Body (JSON):**
  + {
  + "page": 1,
  + "size": 10
  + }
  + content\_copyUse code [with caution](https://support.google.com/legal/answer/13505487).Json
  + **Headers:**
  + Content-Type: application/json
  + X-PIXXI-TOKEN: FjuseQnHvSZy4jTqs8EN6uHfRz85YGv- (**This token needs a secure method of access in production.**)
  + **Response (Expected JSON):** A JSON object containing search results (locations). The statusCode shows success (200) or failure. An empty data array indicates no results. Example:
  + {
  + "statusCode": 200,
  + "data": [
  + { "cityId": 123, "regionId": 456, "communityId": 789, "fullName": "Dubai, Jumeirah" },
  + // ...more locations
  + ]
  + }
* API Request 2: Book a Visit (POST)

Request Method: POST

Request URL: https://dataapi.pixxicrm.ae/pixxiapi/webhook/v1/form

Request Body:

{

"agentId": PropertyData.agentId,

"propertyId": propertyId,

"clientSource": "Eplog Properties",

"extraData": { "Subject": data.subject, "Message": data.message },

"formId": "form-id e.g 9ec22cdf-5dac-4a2a-b17a-a50f58af98fb ",

"leadType": Type,

"name": data.name,

"email": data.email,

"phone": data.phone

}

* Headers:
* X-PIXXI-TOKEN: <YOUR\_API\_TOKEN> // Replaced with <YOUR\_API\_TOKEN>

Expected Response: The response is handled based on the response.ok status. A successful response (response.ok === true) is assumed to mean successful submission, while any other status indicates an error. The exact structure of the response is not shown in the code, but it's likely a JSON object containing a status code and possibly a message.

API Request 3: Fetch Property Details (GET)

Request Method: GET

Request URL: https://dataapi.pixxicrm.ae/pixxiapi/v1/${propertyId}

Request Body: None (GET request)

Headers:

Content-Type: application/json

X-PIXXI-TOKEN: <YOUR\_API\_TOKEN> //Replaced with <YOUR\_API\_TOKEN> for security

Expected Response: The response is parsed as JSON. A successful response (DevelopersData.statusCode === 200) contains property details in the DevelopersData.data field. The structure of DevelopersData.data is not specified in the provided code but is assumed to contain various property attributes.

**IV. Case Studies**

* Describe the purpose and usage of each Component and how they interact to get the result. Explain what data is stored in the context and how components access and modify it.

Implementing Searchable Property Listings with Client-Side Filtering and Pagination

This case study examines the implementation of a searchable property listing system. The system allows users to filter properties based on various criteria and paginates the results for efficient display. The primary components involved are HeroSearchSection.jsx (displays properties), SearchBar.jsx (handles user input and filtering), and Test.jsx (a custom search input component).

**Problem:**

The application needed a way to display a large number of properties dynamically, allowing users to refine their search using several filters (developer, property type, number of bedrooms) and navigate through paginated results. The initial implementation likely lacked efficient data handling for large datasets and a user-friendly filtering mechanism. Directly fetching and displaying all properties at once would be inefficient and could lead to performance issues.

**Solution:**

The solution employs several key strategies:

1. **Context API for Data Sharing:** The usePropertyData custom hook manages the application's property search criteria (filters). This allows components like HeroSearchSection and SearchBar to access and modify the search parameters without prop drilling. This centralized data management is crucial for consistency and maintainability.
2. **Server-Side Filtering with Pagination:** HeroSearchSection fetches properties from a remote API (https://dataapi.pixxicrm.ae/pixxiapi/v1/properties/Eplog Properties) using a POST request. This request includes pagination parameters (size and page), ensuring only a subset of data is fetched at a time. The API handles the actual filtering based on the search criteria provided in PropertyData from the context. The AbortController effectively handles cancellation of previous requests when new ones are initiated (e.g., changing the page or filter), preventing race conditions and unnecessary API calls.
3. **Client-Side Filtering for Developers:** While the main filtering is done server-side for efficiency, the SearchBar component fetches a list of developers and offers client-side filtering for developers. This allows for a more immediate user experience when selecting a developer. This client-side approach improves responsiveness for this specific filter, but server-side filtering remains the core approach for other search criteria to handle the larger data set efficiently.
4. **ReactPaginate for Pagination:** The react-paginate library provides a readily available and customizable pagination component. HeroSearchSection uses this to render pagination controls, updating the page state in usePropertyData and triggering a new API request for the next page.
5. **Custom Search Input with Location Autocomplete:** The Test.jsx component provides a more sophisticated autocomplete feature for searching locations. The autocomplete is powered by API calls, allowing the users to search using partial names and receiving suggestions. It also actively updates the InputData state when a location is selected, ensuring this input is included in the search parameters. This greatly enhances the search functionality and makes it more user-friendly.
6. **Error Handling and Loading States:** Both HeroSearchSection and Test.jsx include robust error handling and loading states (loading, fetching, noResults). This ensures a smooth user experience by displaying appropriate messages and loaders during API calls and handling potential errors gracefully.
7. **Efficient State Management in SearchBar:** SearchBar efficiently manages various filter states (developer, property type, bedroom numbers) using useState and useEffect hooks. The handleFilterChange function updates the selectedFilters state, and useEffect hooks ensure that the changes are reflected in the PropertyData in the context, synchronizing the filters with the data fetching in HeroSearchSection.

**Design Choices:**

* The use of the Context API simplifies data sharing.
* Server-side filtering optimizes performance for large datasets.
* Client-side developer filtering improves responsiveness.
* ReactPaginate offers a user-friendly pagination experience.
* Comprehensive error handling and loading states enhance user experience.
* The separation of concerns among components makes the code more maintainable.

**VII. Future Enhancements**

Advanced Filtering and Sorting:

* Problem: The current filtering is limited.
* Solution: Add more sophisticated filtering options such as:
  + Price range sliders: Allow users to specify a price range using sliders.
  + Date range filters: Filter properties based on completion dates or listing dates.
  + Multiple property type selection: Allow users to select multiple property types simultaneously.
  + Sorting options: Allow sorting of properties by price, size, or date.

**. SEO Optimization and Migration to Next.js:**

* **Problem:** The current React application lacks built-in SEO capabilities. Server-side rendering (SSR) is essential for optimal search engine indexing and performance.
* **Solution:** Migrate the application to Next.js. Next.js provides built-in SSR, automatic code splitting, and features that simplify SEO implementation (e.g., <Head/> component for meta tags). This would significantly improve the website's search engine visibility. The existing React components could largely be reused within the Next.js framework.

**. SEO Optimization and Migration to Next.js:**

* **Problem:** The current React application lacks built-in SEO capabilities. Server-side rendering (SSR) is essential for optimal search engine indexing and performance.
* **Solution:** Migrate the application to Next.js. Next.js provides built-in SSR, automatic code splitting, and features that simplify SEO implementation (e.g., <Head/> component for meta tags). This would significantly improve the website's search engine visibility. **The existing React components could largely be reused within the Next.js framework.**

**Interactive Map Integration:**

* **Problem: Currently, location information is textual.**
* **Solution: Integrate an interactive map to visually display property locations. Users could zoom, pan, and filter properties geographically. This would significantly improve the property discovery experience.**