Dream Neighbourhood .com

Dream Neighbourhood

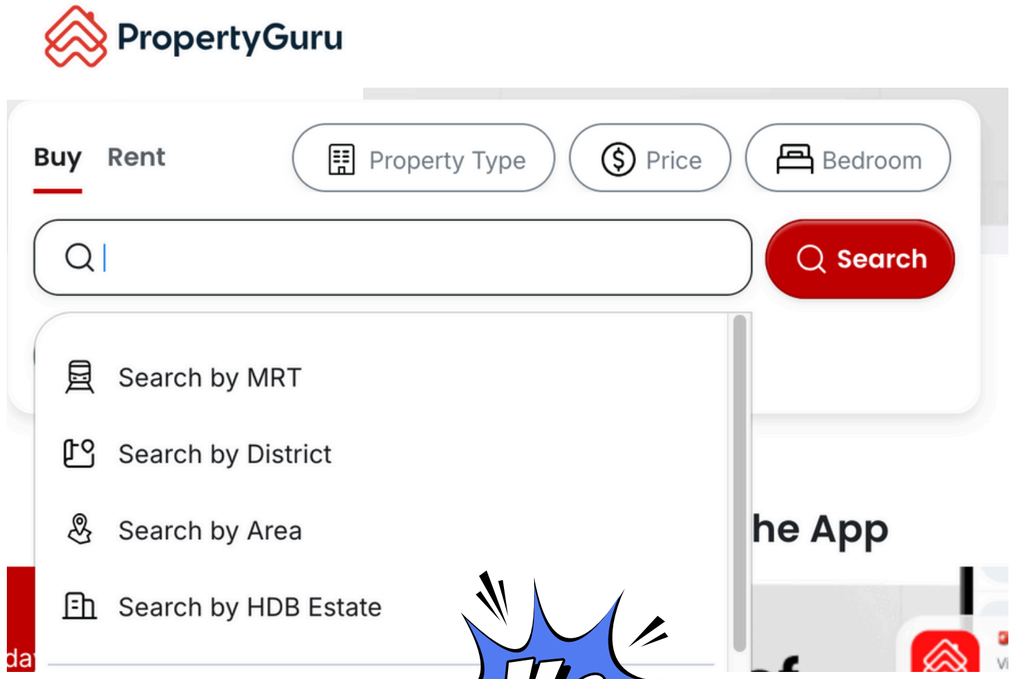
By :

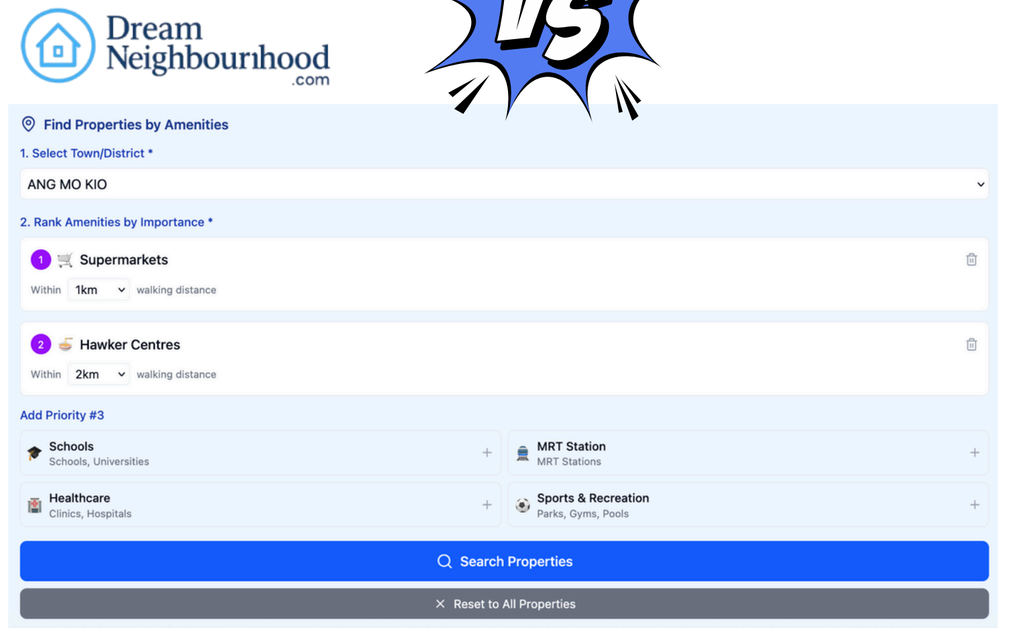
- How En Hsin
- Veda Ho Yong Qian
- Jarrel Ng Tze Xuan
- Isaac Teo Yi Jay
- Sean Winston Susanto
- Joel Ng Wei Heng

Explore Now

Problem Statement

- Finding the right home and neighbourhood in Singapore is difficult since existing platforms focus mainly on listings and prices, not lifestyle fit.
- Home seekers struggle to compare areas based on amenities, accessibility, and community vibe.
- There’s no centralised platform that helps users discover neighbourhoods suited to their daily routines, priorities, and long-term needs.

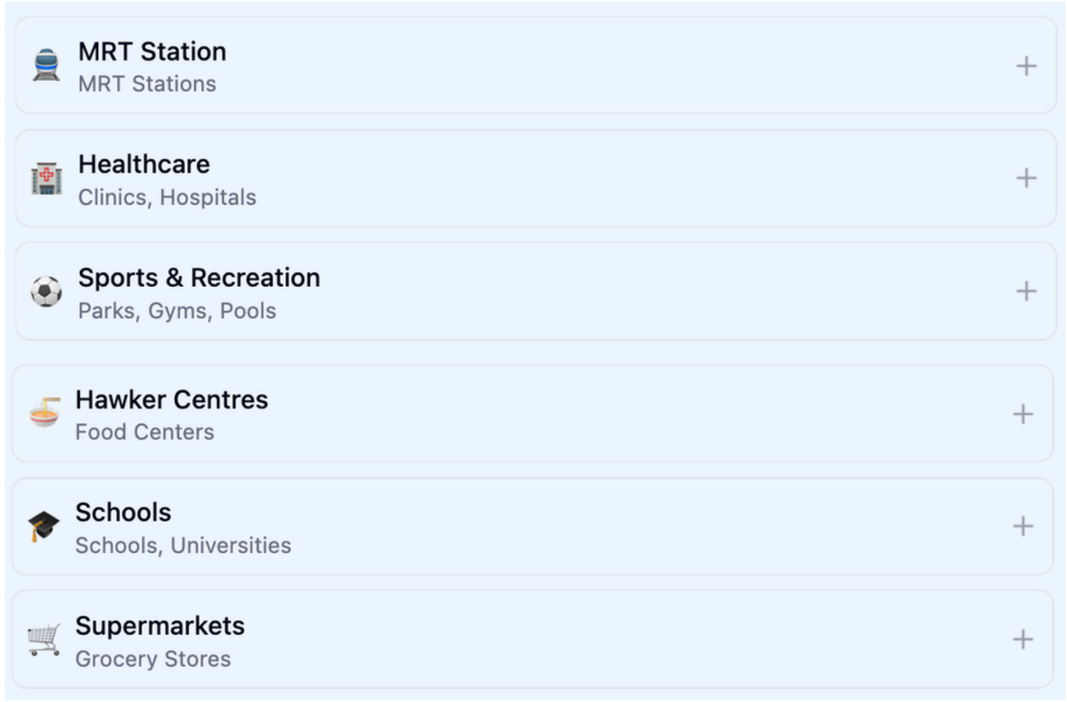




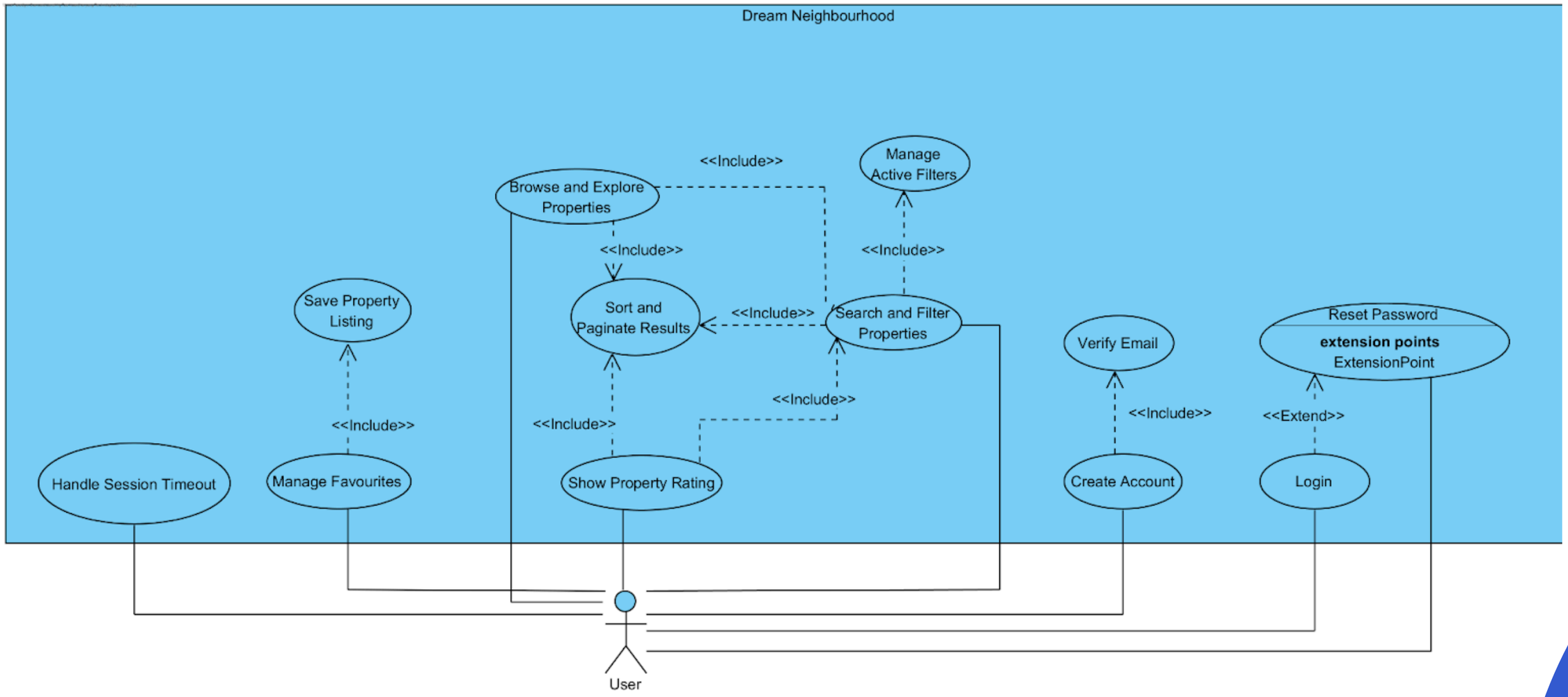
Target Users

- Home seekers in Singapore — individuals or families looking for homes that match their lifestyle and budget.
- Includes young professionals, expats, families and retirees.
- Users who want to find not just a house, but a neighbourhood that fits their daily life.

Search by amenities nearby in the neighbourhood



Use Case Diagram



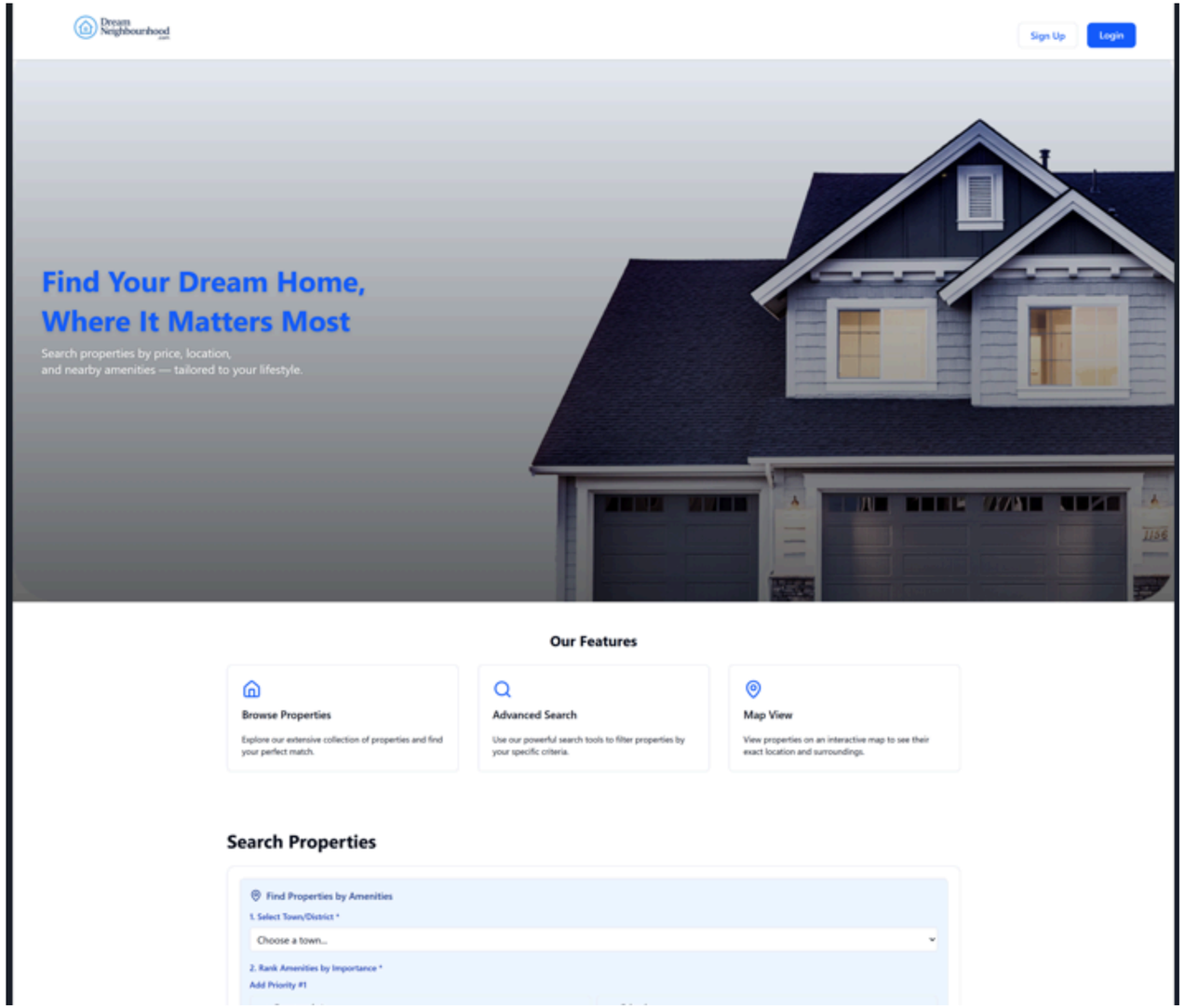
External APIs

- **Resale Flat Prices Dataset API**
  - real HDB resale data as our property dataset
- **Geoapify Places API**
  - Identify nearby amenities
  - Retrieve distance from amenities to rank properties
- **OpenStreetMap Tile API from Leaflet library**
  - Render an interactive map and location markers
- **Supabase (PostgreSQL as a Service)**
  - Accessed via Supabase’s REST/DB API using project URL + service role key
  - Stores Property data (HDB flats), Users (auth, verification, reset tokens) and User favourites (Saved listings)

Tech Stack

Frontend	Backend
Framework: React + Vite	Framework: Express.js
Language: Typescript	Runtime: Node.js
Styling: Tailwind CSS	Database Layer: Supabase (PostgreSQL) via Supabase client / REST

DEMO


Dream Neighbourhood
Sign Up
Login
Find Your Dream Home,
Where It Matters Most
Search properties by price, location,
and nearby amenities — tailored to your lifestyle.
Our Features
Browse Properties
Explore our extensive collection of properties and find your perfect match.
Advanced Search
Use our powerful search tools to filter properties by your specific criteria.
Map View
View properties on an interactive map to see their exact location and surroundings.
Search Properties
Find Properties by Amenities
1. Select Town/District *
Choose a town...
2. Rank Amenities by Importance *
Add Priority #1

SWE Practices & Design patterns

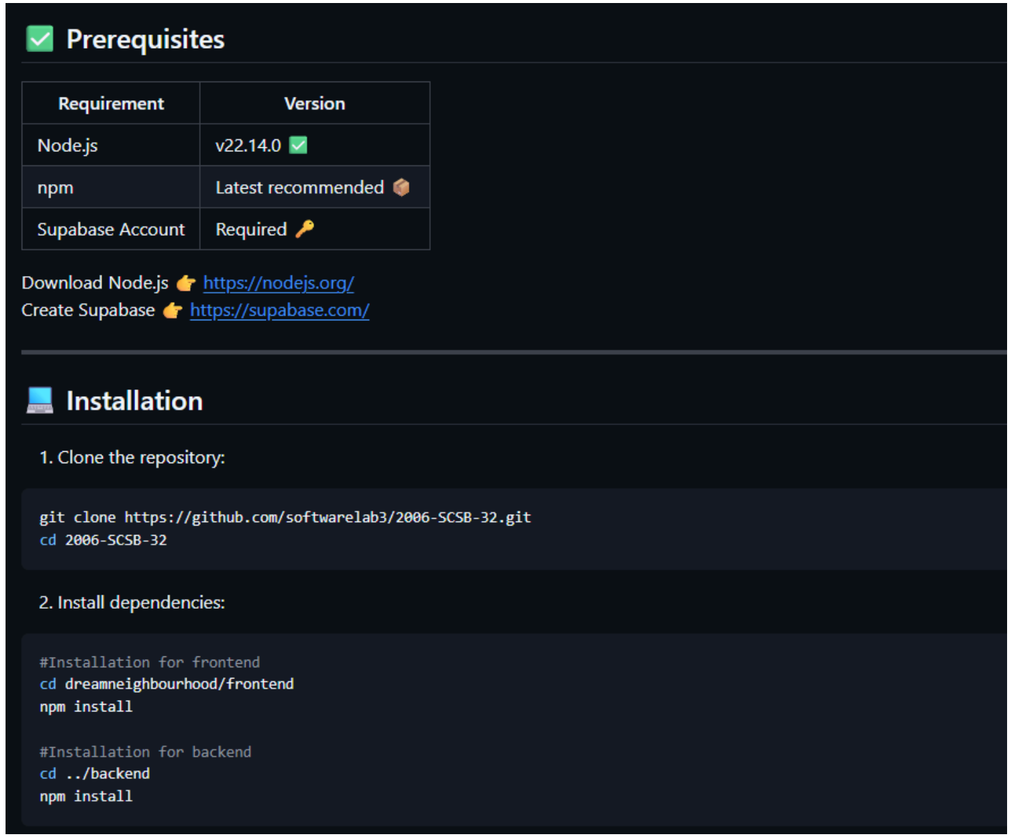
Find your Dream Home, where it matters most

Documentation - README

🏠 Dream Neighbourhood — 2006-SCSB-32

This project helps home seekers discover properties tailored to their lifestyle.
Users rank the amenities that matter most to them, and the system calculates scores for each property based on proximity to these ranked amenities.
The best-matched homes are then displayed in sorted order — a personalized and data-driven home recommendation experience.

📋 Table of Contents



✅ Prerequisites

Requirement	Version
Node.js	v22.14.0 ✅
npm	Latest recommended 📦
Supabase Account	Required 🔑

Download Node.js 👉 https://nodejs.org/
Create Supabase 👉 https://supabase.com/

💻 Installation

1. Clone the repository:

```
git clone https://github.com/softwarelab3/2006-SCSB-32.git
cd 2006-SCSB-32
```

2. Install dependencies:

```
#Installation for frontend
cd dreamneighbourhood/frontend
npm install

#Installation for backend
cd ../backend
npm install
```

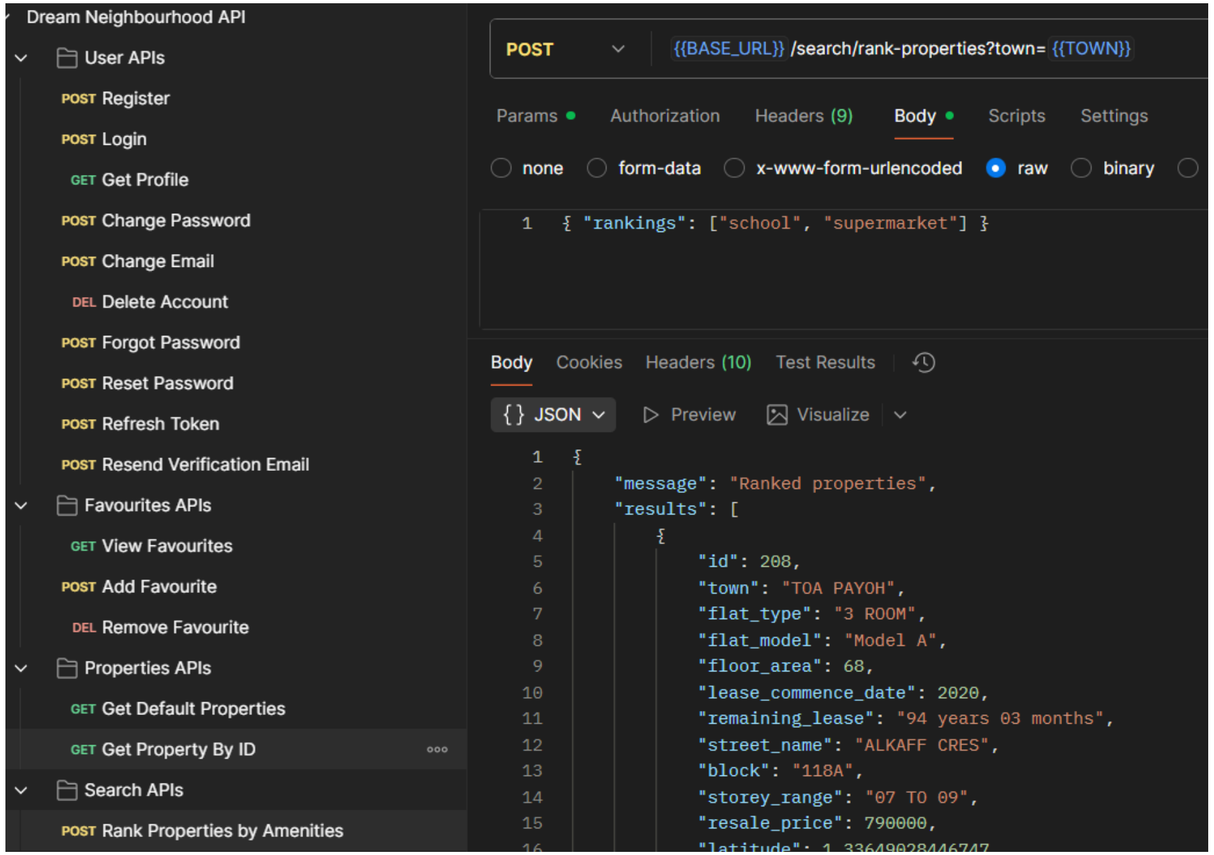
Explains project purpose, setup, and usage for new developers and serves as a reference for updates, APIs, and workflows.

Documentation – API DOCS

📌 API Documentation

All endpoints are prefixed with `/api/v1`. Protected routes require authentication (`authenticateToken`).

Method	Endpoint	Description	Request Body / Query Parameters
Users			
POST	`/api/v1/users/register`	Register a new user	`{ "username": "...", "email": "...", "password": "..." }`
GET	`/api/v1/users/verify-email`	Verify user email	Query param: `token=...`
POST	`/api/v1/users/login`	Login a user	`{ "email": "...", "password": "..." }`
POST	`/api/v1/users/forgot-password`	Request password reset	`{ "email": "..." }`
POST	`/api/v1/users/reset-password`	Reset password using token	`{ "token": "...", "newPassword": "..." }`
GET	`/api/v1/users/profile`	Get authenticated user profile	Protected (Bearer token)
POST	`/api/v1/users/change-password`	Change user password	`{ "oldPassword": "...", "newPassword": "..." }` Protected
POST	`/api/v1/users/change-email`	Change user email	`{ "newEmail": "..." }` Protected
DELETE	`/api/v1/users/delete-account`	Delete user account	Protected
POST	`/api/v1/users/refresh-token`	Refresh JWT token	`{ "refreshToken": "..." }` Protected
POST	`/api/v1/users/send-verification-email`	Resend verification email	Protected
Favourites			
GET	`/api/v1/favourites/`	View all favourite properties for authenticated user	Protected
POST	`/api/v1/favourites/add`	Add a property to favourites	`{ "propertyId": 1 }` Protected

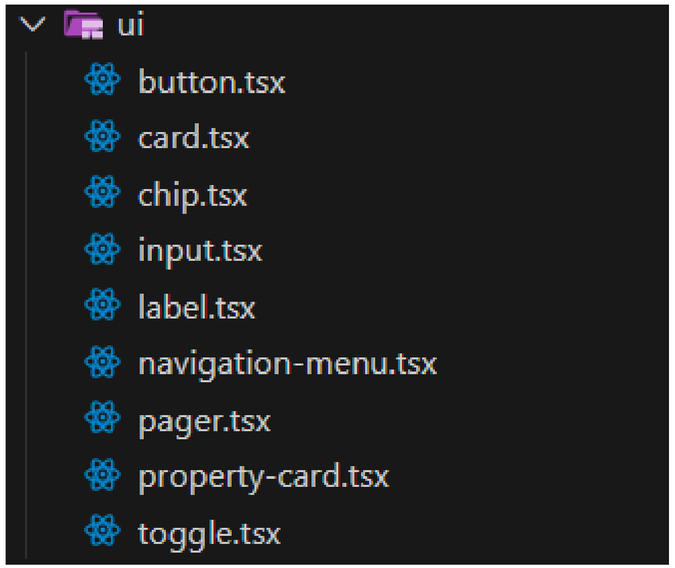


Postman collections and API docs let developers test, understand, and integrate APIs quickly while maintaining consistency and collaboration

Good Code Practices

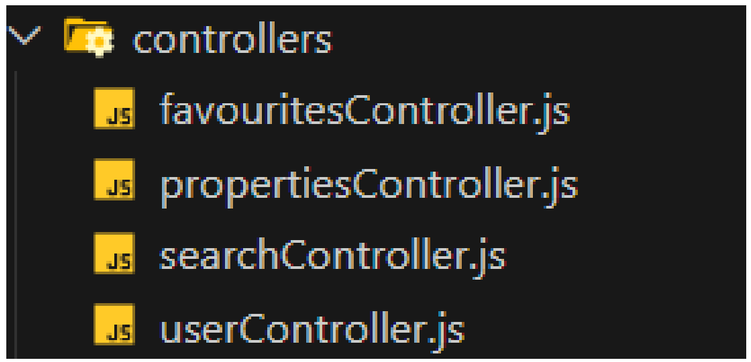
Modular & Reusable Code

- Using consistent UI Components from Shadcn library
- Ensures consistent design across the application
- Avoid creating duplicate components with the same functionality
- Maintainable and Scalable



Code Readability & Organisation

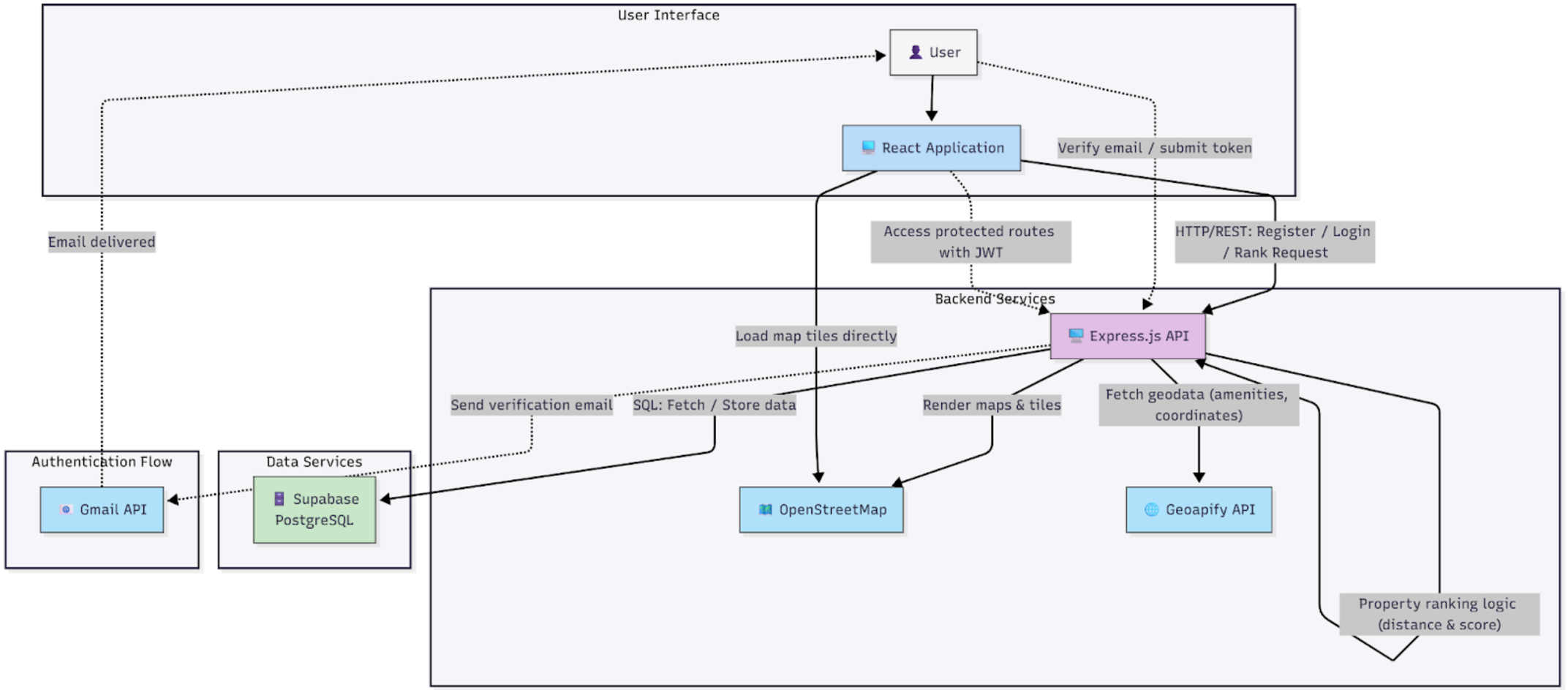
- Meaningful variable, function, and class names
- Consistent file naming convention



System Design

Find your Dream Home, where it matters most

Architecture Diagrams

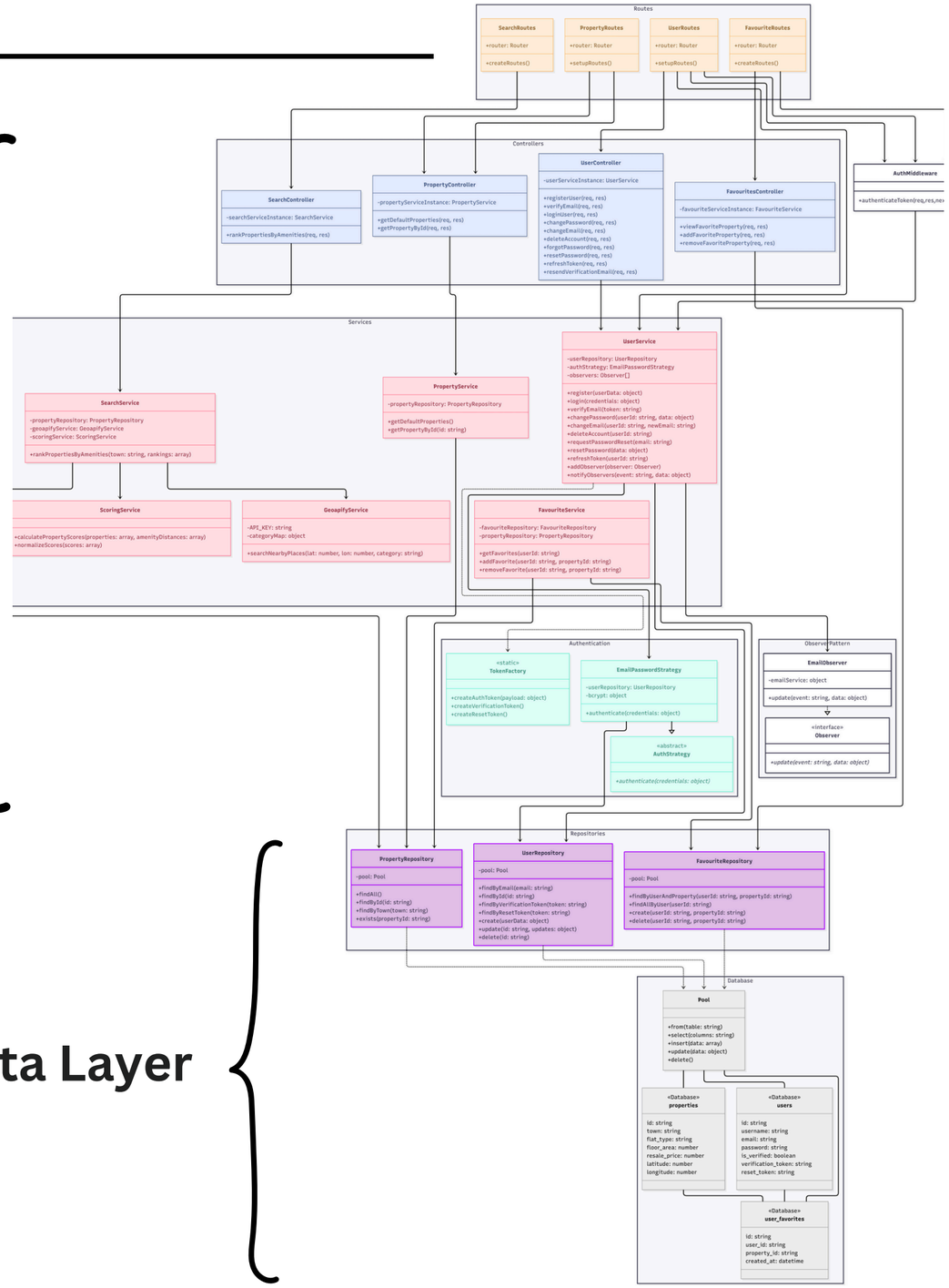


Class Diagram

App Boundary

Logic Layer

Data Layer



Design Patterns

Find your Dream Home, where it matters most

Facade Pattern

```
// services/UserService.js
export class UserService {
  async register(userData) {
    const existingUser = await this.userRepository.findByEmail(userData.email);
    const hashedPassword = await bcrypt.hash(userData.password, 10);
    const verificationToken = TokenFactory.createVerificationToken();
    const newUser = await this.userRepository.create({
      ...userData, hashedPassword, verificationToken
    });
    await this.notifyObservers('USER_REGISTERED', {...});
    return newUser;
  }
}
```

Definition:
Provides a unified interface to a set of interfaces in a subsystem.

Where:
Controllers only call UserService.register() → internally it handles Repository (DB), Factory (token), and Observer (email)

Consequences

- Simplifies controller logic: one call instead of multiple.
- Reduces dependencies between modules.
- Improves maintainability and readability.

Repository Pattern

```
// repositories/UserRepository.js
export class UserRepository {
  async findByEmail(email) {
    const { data, error } = await
pool.from("users").select("*").eq("email", email);
    if (error) throw error;
    return data?.[0] || null;
  }
  async create(userData) {
    const { data, error } = await
pool.from("users").insert([userData]).select().single();
    if (error) throw error;
    return data;
  }
}
```

Definition:
Handles all database operations in one place

Where:
UserService simply calls repository methods instead of writing queries itself

Consequences

- Single Responsibility
- Easier Maintenance: if the database changes (Supabase → MongoDB), only repositories need updating.
- Better Testability
- Cleaner Architecture

Factory Pattern

```
// factories/TokenFactory.js
export class TokenFactory {
  static createAuthToken(payload) {
    return jwt.sign(payload, process.env.JWT_SECRET, { expiresIn: "2h" });
  }

  static createVerificationToken() {
    return crypto.randomBytes(32).toString("hex");
  }
}
```

Definition:
Centralizes and standardizes object creation logic, ensuring consistency and reducing code duplication.

Where:
TokenFactory class responsible for generating tokens in a single location

Consequences

- Consistency: All tokens follow a uniform structure.
- Single Responsibility: Only one class manages token creation.
- Easy Maintenance: Modify token behavior (expiry, format) in one place.
- Code Reuse: Avoids duplication across controllers and services.

Observer Pattern

```
// observers/EmailObserver.js
export class EmailObserver {
  constructor(emailService) { this.emailService = emailService; }
  async update(event, data) {
    if (event === 'USER_REGISTERED')
      await this.emailService.sendVerificationEmail(data.email, data.username, data.token);
  }
}
// services/UserService.js
export class UserService {
  constructor() { this.observers = []; }
  addObserver(observer) { this.observers.push(observer); }
  async notifyObservers(event, data) {
    for (const obs of this.observers) await obs.update(event, data);
  }
  async register(userData) {
    // registration logic...
    await this.notifyObservers('USER_REGISTERED', {...});
  }
}
```

Definition:
Defines a one-to-many dependency so when one object's state changes, all its dependents are notified automatically.

Where:
Implement an Observer mechanism where UserService acts as the Subject, and EmailObserver (and others like SmsObserver) act as Observers that react to events

Consequences

- Decouples user logic from notification logic.
- Easily extendable — add new observers (SMS, analytics, etc.) without touching UserService.

Singleton Pattern

```
// controllers/userController.js
let userServiceInstance = null;

function getUserService() {
  if (!userServiceInstance) {
    userServiceInstance = new UserService();
    const emailObserver = new EmailObserver(emailService);
    userServiceInstance.addObserver(emailObserver);
  }
  return userServiceInstance;
}
```

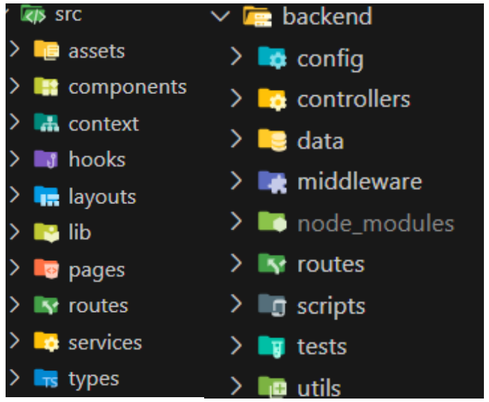
Definition:
Ensures that only one instance of a class exists throughout the application and provides a global point of access to it

Where:
Use a Singleton approach for the UserService — it's created only once, and all controllers share the same instance.

Consequences

- Consistency: Single shared service instance across controllers.
- Efficiency: Prevents repeated setup (e.g., multiple observers).
- Ease of Access: Global access point for UserService.
- Thread-Safe Simplicity: Ideal for small Node.js backends.

SOLID Principles



Single Resposibility Principle (SRP)

Each package has a distinct responsibility, and the specific classes within a package focus on a particular business model or a cohesive set of related logic.

Open-Closed Principle (OCP)

By using factory and singleton patterns, the application can be extended with new functionality without modifying existing code, keeping it open for extension but closed for modification.

Dependency Inversion Principle (DIP)

Controllers depend on abstractions like the database pool and email utilities instead of concrete implementations. This makes the code loosely coupled, easier to maintain, and allows swapping or mocking services without changing high-level logic

Interface Segregation Principle (ISP)

Each service such as authentication, favourites, and properties, has its own specific interface or controller, keeping components independent and prevents classes from depending on methods they don’t use

Traceability

Search, Filter & Rank Properties

Find your Dream Home, where it matters most

Search, Filter & Rank Properties -

Use Case Description

requirement → **use case** → dialog flow → design → testing

Actor	User
Description	Allows Users to search for properties with filters
Preconditions	User is logged in Property listings database and map/geolocation service is available
Postconditions	System displays filtered and sorted property listings
Priority	High
Frequency of Use	High
Flow of Events	1. User navigates to the search interface 2. User enters a Town 3. User ranks the 6 amenity filters (minimum 1, maximum 6) 4. User specifies the maximum acceptable distance for each amenity (default 1km) 5. User can enter one key locations (postal code or place name) 6. System validates the location input and calculates distance 7. User can use the slider to enter a minimum budget 8. System validates the range 9. User can select minimum number of bedrooms 10. User can use the slider to enter a minimum square area 11. User clicks the search button 12. System displays results in pages of 20 items 13. User can clear filters at any time
Alternative Flows	1. User clears filters, system will reset and shows all listings 2. User modifies filters (e.g., budget, amenities), system dynamically updates the result 3. User removes specific key locations or amenities,
Exceptions	1. Network or database failure, the system will display an error message. 2. No listings are found, and the system displays, “No properties found.” 3. Invalid budget input (e.g., min > max), the system displays, “minimum amount cannot be more than the maximum” 4. Distance calculation fails due to missing map data; the system will exclude affected listings from results
Includes	Manage Active Filters Sort and Paginate Results
Special Requirements	Filters must update results dynamically while filters are modified System must support up to 5 key locations and 10 amenities per search Response time for applying filters should not exceed 3 seconds
Assumptions	The property database is available Property listings contain valid postal codes and coordinates
Notes and Issues	UX challenge when filters conflict and eliminate all listings Handling duplicate or ambiguous amenity keywords (e.g., “school” vs “Primary school” Performance may degrade when many filters are applied simultaneously

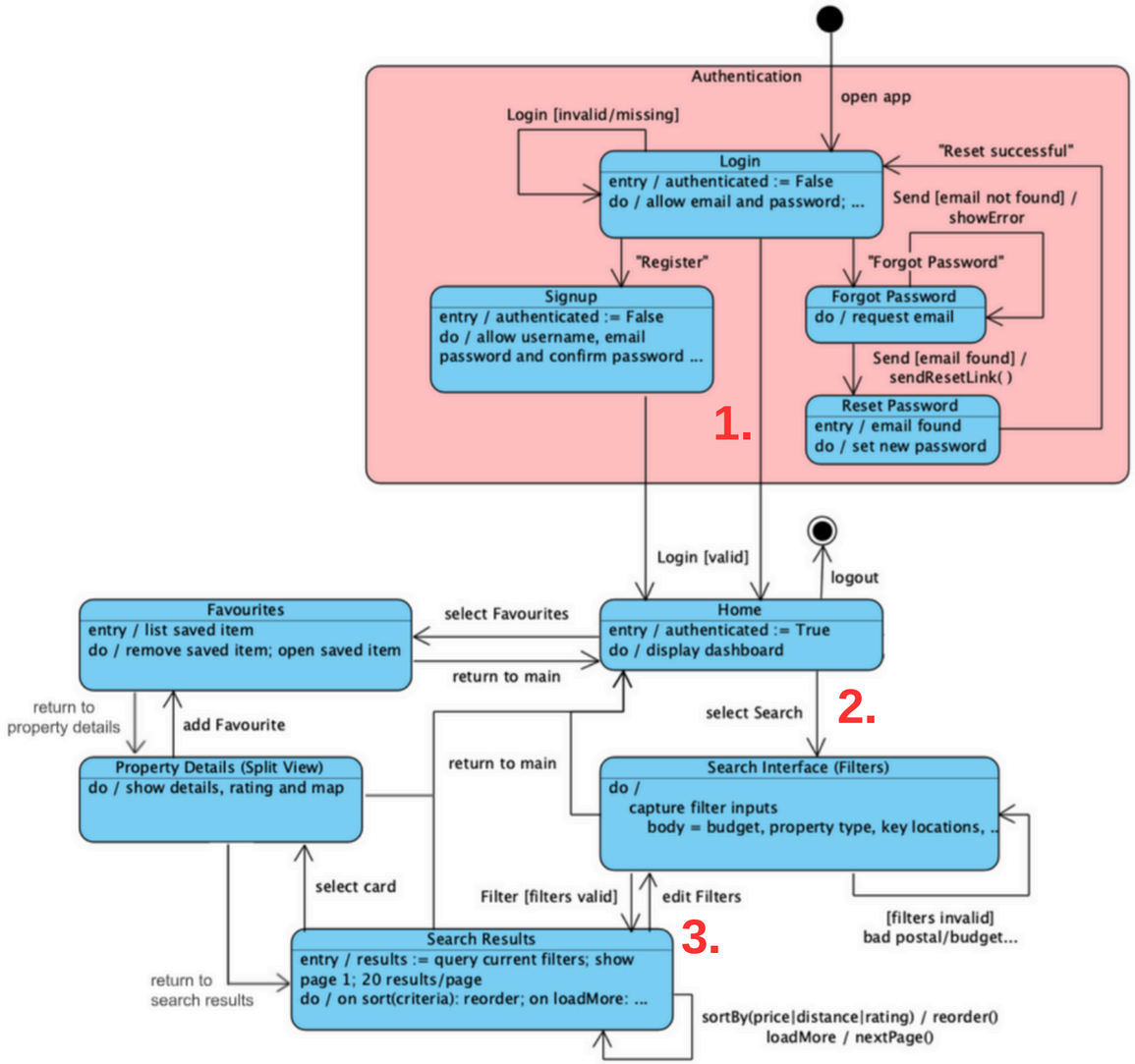
Search, Filter & Rank Properties -

Dialog Map

1. The user logs in and arrives at the **Home** page.

2. They choose **Search** and go to the **Search Interface (Filters)** state. In this state, they will set the town, and then click to prioritise at least 1 of the 6 amenity groups and can optionally choose to set price, rooms and area.

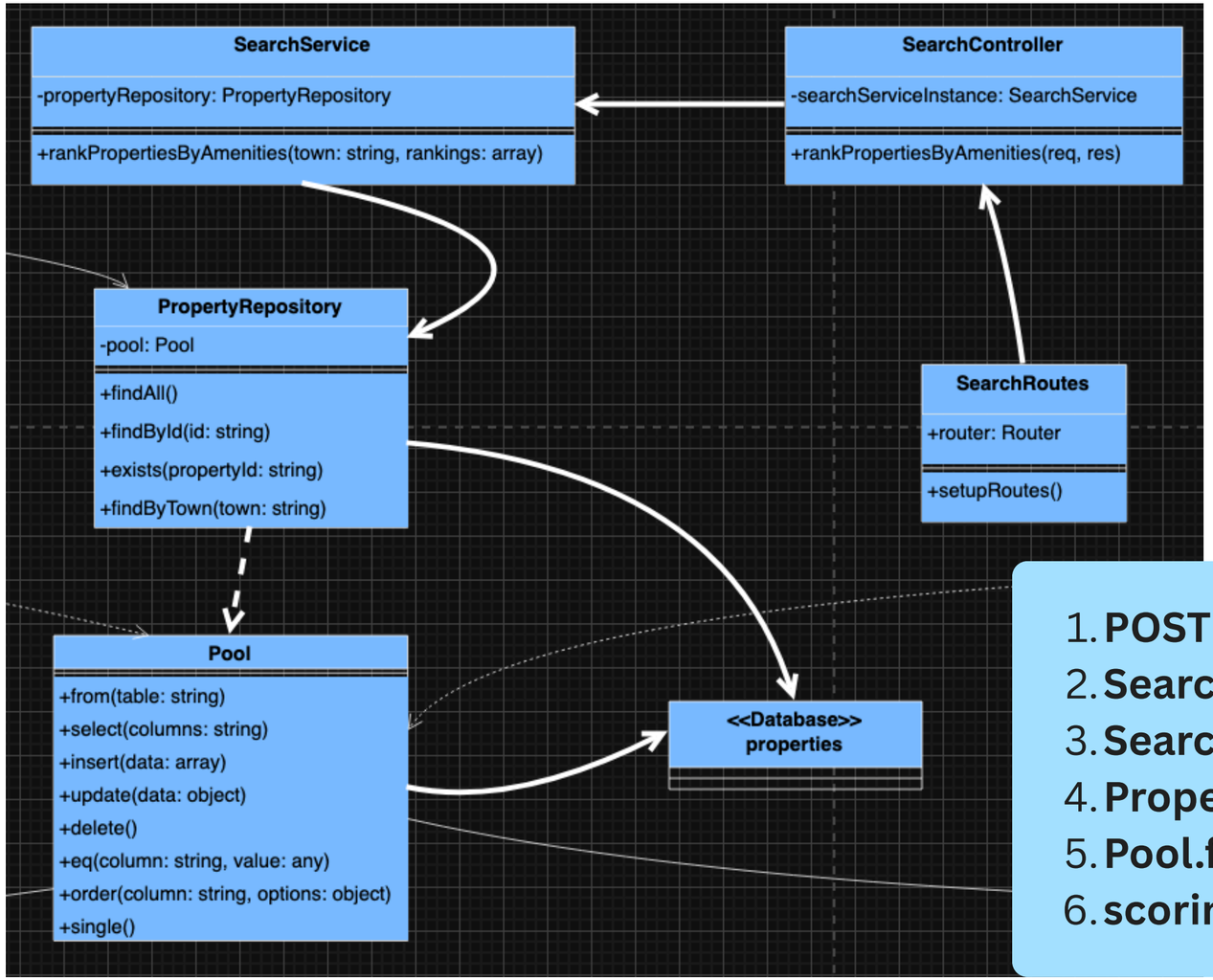
3. When they hit Search, the app transitions into **Search Results**, showing a ranked list of properties.



Search, Filter & Rank Properties –

Class Diagram

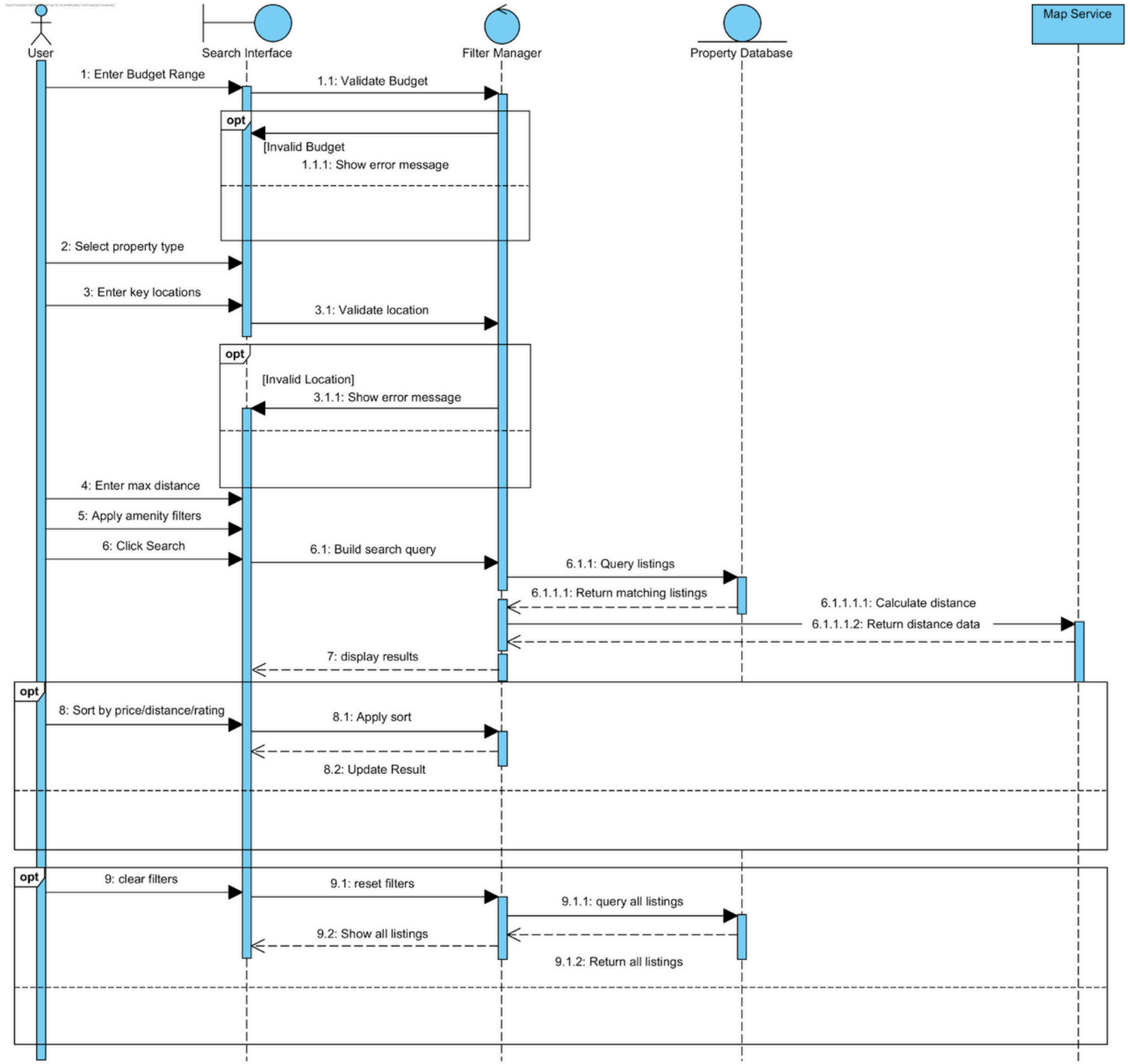
requirement → use case → dialog flow → **design** → testing



1. **POST /api/v1/search/rank-properties**
2. **SearchController.rankPropertiesByAmenities()**
3. **SearchService.rankPropertiesByAmenities(town, rankings)**
4. **PropertyRepository.findByTown(town)**
5. **Pool.from('properties').select('*').eq('town', town)**
6. **scoringService.scoreProperty(...)**

Search, Filter & Rank Properties -

Sequence Diagram



Search, Filter & Rank Properties –

Good Design Practices

requirement → use case → dialog flow → **design** → testing

Traceability → method names match diagrams & route (POST /search/rank-properties)

Facade in Controller → orchestration only; no SQL/business logic

```
// controllers/searchController.js
export const rankPropertiesByAmenities = async (req, res) => {
  try {
    const { rankings } = req.body;
    const { town } = req.query;
    const result = await getSearchService()
      .rankPropertiesByAmenities(town, rankings);
    return res.status(200).json(result);
  } catch (err) {
    if (err.message === "town is required")           return res.status(400).json({ message: err.message });
    if (err.message === "rankings array is required in body") return res.status(400).json({ message: err.message });
    if (err.message === "No properties found")         return res.status(404).json({ message: err.message });
    return res.status(500).json({ message: "Server error", error: err.message });
  }
};
```

Controller = thin facade (parse → call service → map errors)

Search, Filter & Rank Properties -

Good Design Practices

requirement → use case → dialog flow → **design** → testing

Business rules in Service → validation, scoring, sorting, pagination

```
// services/searchService.js
export class SearchService {
  async rankPropertiesByAmenities(town, rankings) {
    if (!town) throw new Error("town is required");
    if (!rankings || !Array.isArray(rankings) || rankings.length === 0)
      throw new Error("rankings array is required in body");

    const properties = await this.propertyRepository.findByTown(town.toUpperCase());
    if (!properties || properties.length === 0) throw new Error("No properties found");

    const scored = await Promise.all(
      properties.map(p =>
        this.scoringService.scoreProperty(
          p, rankings, this.geoapifyService.fetchAmenitiesAroundProperty.bind(this.geoapifyService)
        )
      )
    );

    scored.sort((a, b) => b.totalScore - a.totalScore);        // desc by score
    return { message: "Ranked properties", results: scored };
  }
}
```

Service owns rules (defensive checks, pure scoring, sorted results)

Repository pattern → parameterized DB access via Pool (injection-safe)

```
// repositories/propertyRepository.js
export class PropertyRepository {
  async findByTown(town) {
    const { data, error } = await pool      // pool is your gateway client
      .from("properties")
      .select("*")
      .eq("town", town);                    // parameterized equality
    if (error) throw error;
    return data || [];
  }
}
```

Repository pattern (no business logic; safe, testable DB calls)

Search, Filter & Rank Properties –

Tests

On the backend we wrote **black-box tests** for the */api/v1/search/rank-properties* endpoint and the scoring function.

Our test cases include:

- valid searches with different amenity rankings and towns,
- boundary cases like zero amenities selected, very high or very low budgets,
- and invalid inputs such as missing town, negative distances or out-of-range ranks.

For each test we check that the response status is correct, that the list is sorted by score in descending order, and that invalid inputs return clear error messages instead of crashing.

Future Plans

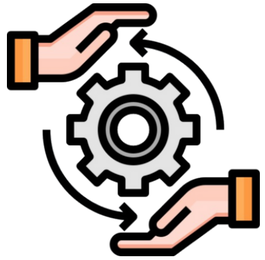
Future Expansion

Real-time property updates

- Integrate Supabase Realtime so users can see live listings & price changes

Recommendation System

- Implement machine-learning model to suggest properties

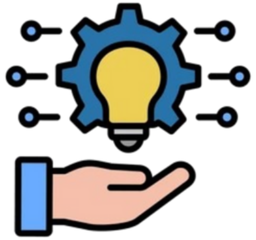


Technical Improvements

- Microservice architecture: split modules (auth, search, maps) for better scalability and independent deployment.
- Caching layer (Redis): improve search performance by caching frequent queries.

UI/UX Enhancements

- Add interactive maps with heat-zones for property prices.
- Improve mobile responsiveness and dark-mode support.
- Provide multi-language support for wider accessibility.



Future Design Pattern Adoption

- Observer Pattern: for push notifications and real-time updates when favourites change.
- Factory Pattern: for creating different user types (buyer, seller, admin) with distinct privileges.

Thank You

Find your Dream Home, where it matters most