University of Pisa

Artificial Intelligence and Data Engineering

Large Scale and Multi-Structured Databases

HomeXplore

Projectual Documentation

Group Members:

Alessandro Lagonegro
Ferdinando Muraca
Carlo Vincenzo Stanzione

Academic Year - 2024/2025

Abstract

"Are you looking for the house of your dreams?" That's the motto for most real estate online marketplaces. But HomeXplore keeps your feet on the ground. We provide more than dreams; we give you the insights to find the perfect reality.

Imagine not just picturing your dream home but understanding how it fits into your everyday life. Knowing how close the nearest school is and where the green areas are isn't just useful; it's essential.

With HomeXplore, you're equipped with these critical geographic insights, empowering you to make a fully informed choice. Our platform benefits sellers too, offering unparalleled exposure and genuine connections with informed buyers. The online experience closely mirrors the real thing, with the convenience of booking in-person viewings right from our site.

The code for this project is available on GitHub at:

- https://github.com/effemuraca/homexplore-back
- https://github.com/VinStan1/homexplore-front

Contents

Chapter 1

Introduction

HomeXplore is a user-friendly platform designed to facilitate property sales, serving as a meeting point between sellers and buyers. The application aims to simplify, accelerate, and enhance the home-buying process by providing advanced search filters, an interactive map for exploring surrounding areas, a booking system for scheduling viewings, and access to market analytics.

For sellers, *HomeXplore* offers a powerful tool to reach a wide audience of potential buyers. The booking system plays a crucial role in fostering communication between buyers and sellers, enabling the organization of open house events and private viewings. Additionally, real estate agencies can monitor sales performance and efficiently manage their property listings.

Overall, the platform is designed to optimize and streamline the home buying and selling experience for all users.

1.1 Key Features

The main features of the application include:

- **Advanced Search**: Users can browse through thousands of property listings using detailed filters and save their favorite homes for future reference.
- **Interactive Exploration**: Users can view points of interest around a property, such as universities, parks, and hospitals, as well as receive recommendations for similar homes in the area.
- **Booking System**: This feature facilitates direct interaction between sellers and buyers. Sellers can create open house events for property viewings, which buyers can book in advance.
- **Market Analytics**: The platform provides insights into the real estate market, enabling buyers to make informed decisions and allowing sellers to track the

performance of their listings.

Chapter 2

User Manual & Mockups

Upon accessing the website, users are greeted by the presentation page. Here, unregistered users can click the *Sign Up* button, while returning users can *Log In* to access their personalized dashboard. Alternatively, by clicking the *Get Started* button, users are taken directly to the property search page without needing to register.

2.1 House Search Page

On the house search page, users can explore available properties using various filters. Each listing provides a brief preview of the property. Additionally, the top right corner features the *Show Analytics* button for registered users, offering insights into market trends and property prices.

2.2 Property Details

After performing a search, results appear at the bottom of the page. Clicking on a listing opens the detailed property page, where users can view comprehensive information (description, photos, number of bathrooms/bedrooms, etc.). Additional options include:

- *Add to Favourites* – Available only for buyers.
- *See it on the Map* – Accessible to all users.
- *Book Now* – Available for buyers when applicable.

2.3 Geo-Exploration and Map Features

The geo-exploration feature allows users to view the property on an interactive map and explore its surroundings. Key elements on this page include:

- The liveability score (assessing the quality of the location).
- Recommendations for similar houses.
- Information about the city and neighbourhood.

2.4 User Profiles and Dashboards

2.4.1 Buyers

Registered buyers can access their dashboard via the profile icon in the navigation bar. The buyer's dashboard offers:

- *Favourite Properties* – List of saved properties.
- *My Reservations* – Overview of scheduled viewings.
- *Edit Profile* – Option to update personal details.

2.4.2 Sellers

The seller's dashboard provides tools for managing property listings. Main sections include:

- *Properties for Sale* – Manage active listings.
- *Sold Properties* – View details of sold properties.
- *Add Property* – Create a new listing.
- *Edit Profile* – Update personal information.

Within the *Properties for Sale* section, sellers can:

- *Edit* – Modify property details.
- *View Reservations* – Check reservation received.
- *Sell* – Mark the property as sold.
- *Remove* – Delete the listing.

In the *Sold Properties* section, detailed analytics and statistics (including sale dates) are provided to help sellers evaluate their performance.

2.5 Figures

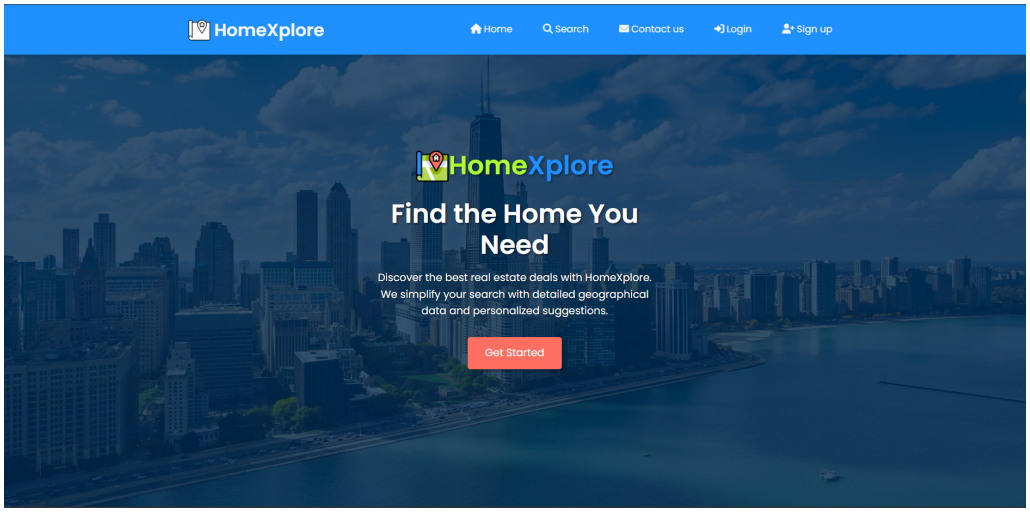

Figure 2.1: Presentation Page (main.html)

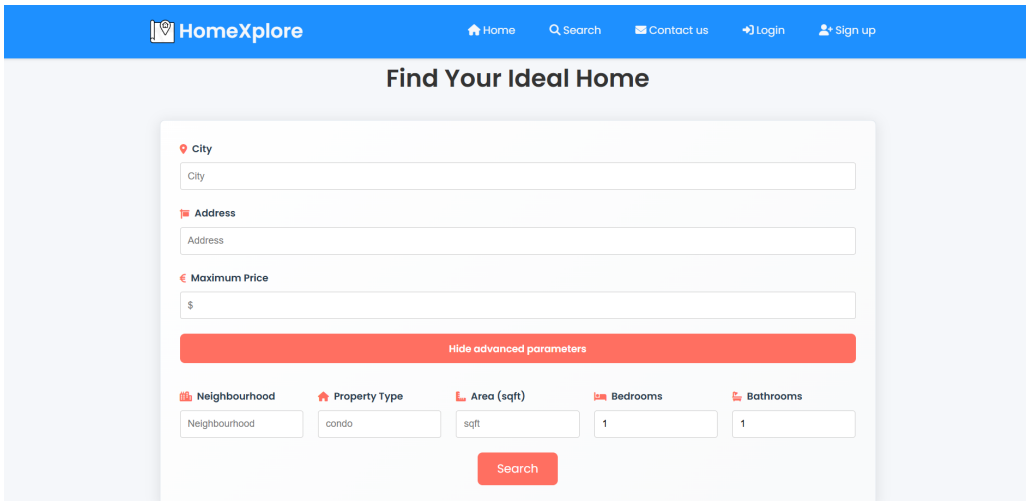

Figure 2.2: House Search Page (index.html) - Request

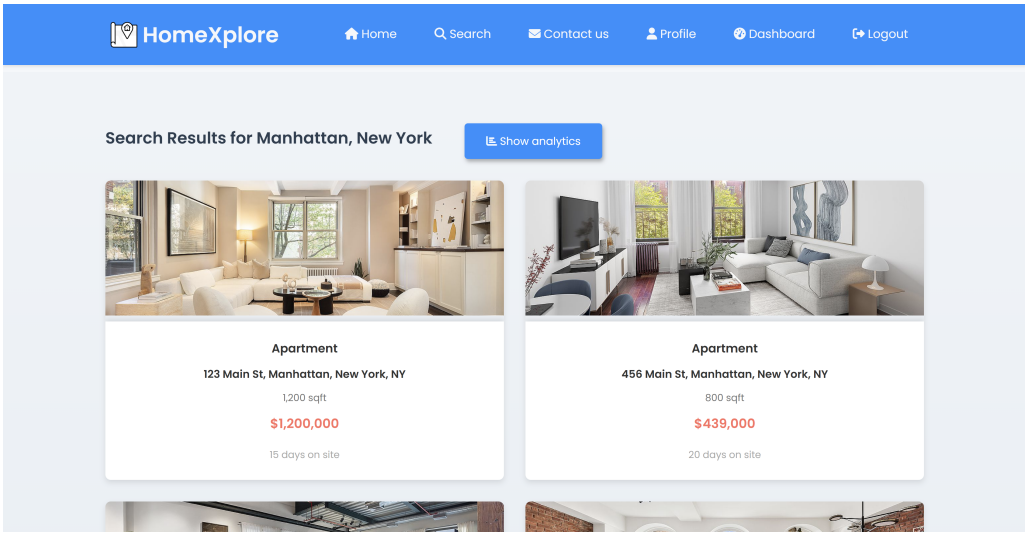

Figure 2.3: House Search Page (index.html) - Response

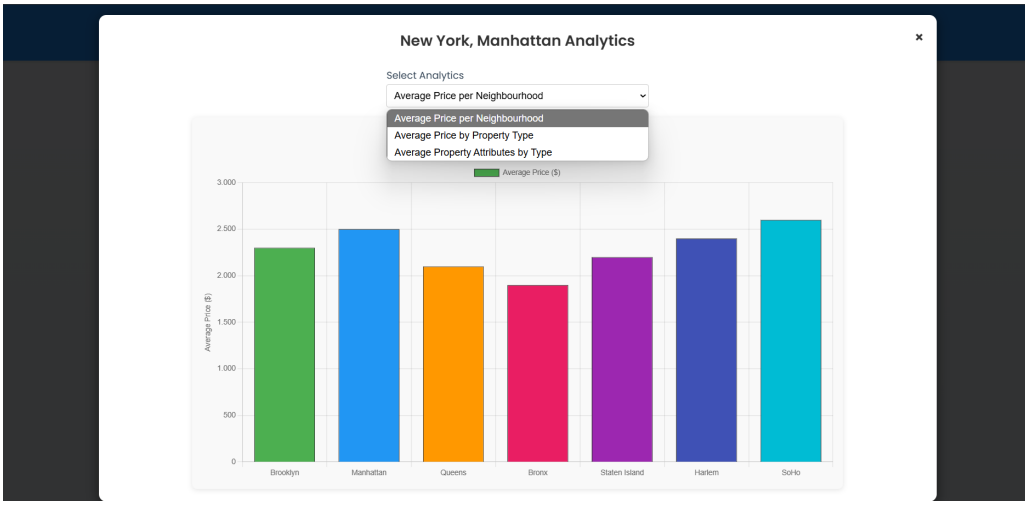

Figure 2.4: Market Analytics Access on the Search Page (index.html)

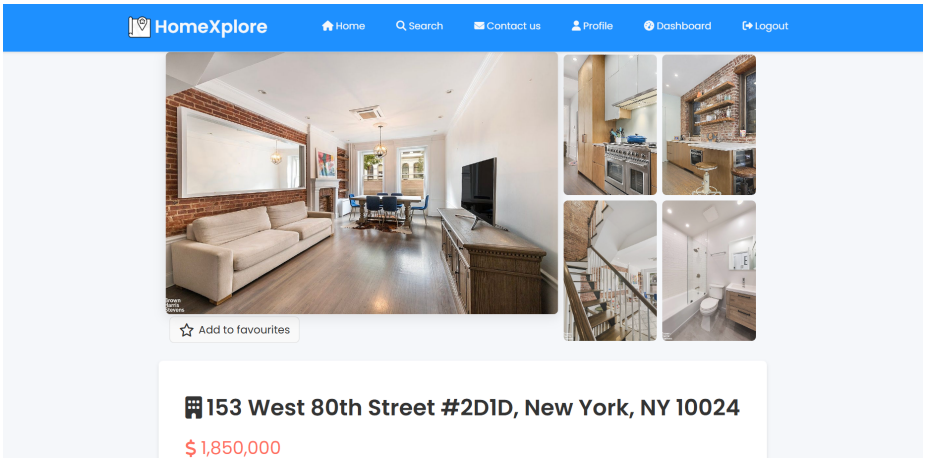

Figure 2.5: Detailed Property View (detail_property.html) - Part 1

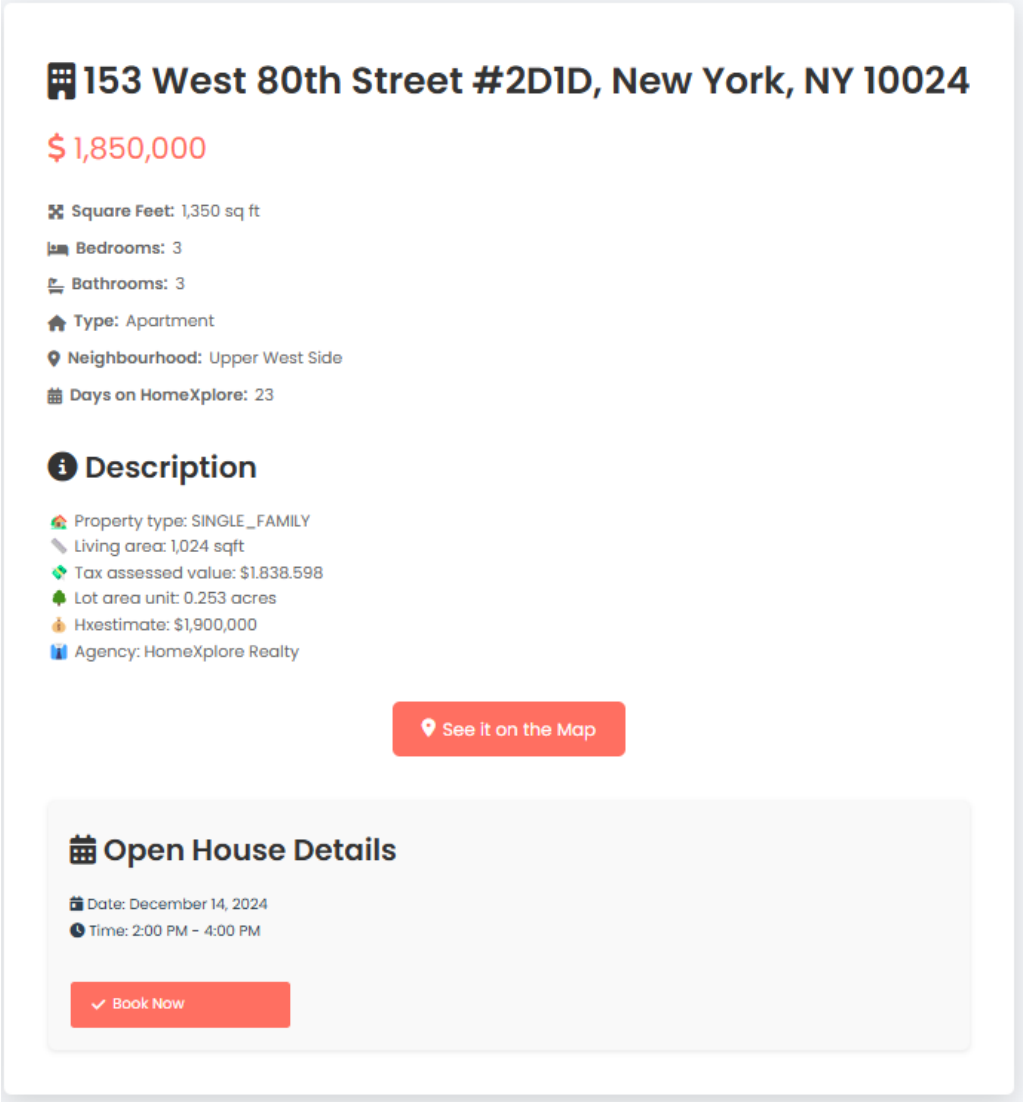

Figure 2.6: Detailed Property View (detail_property.html) - Part 2

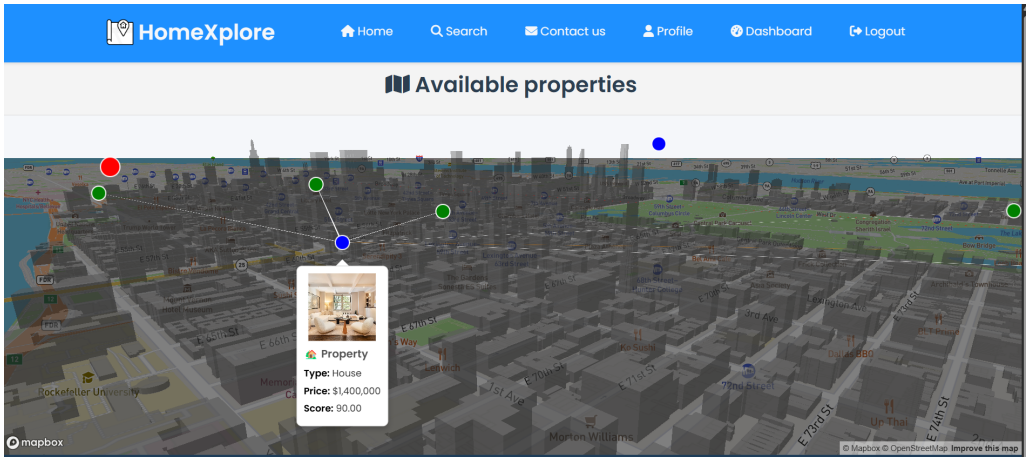

Figure 2.7: Interactive Map Feature (map.html) - Map View 1

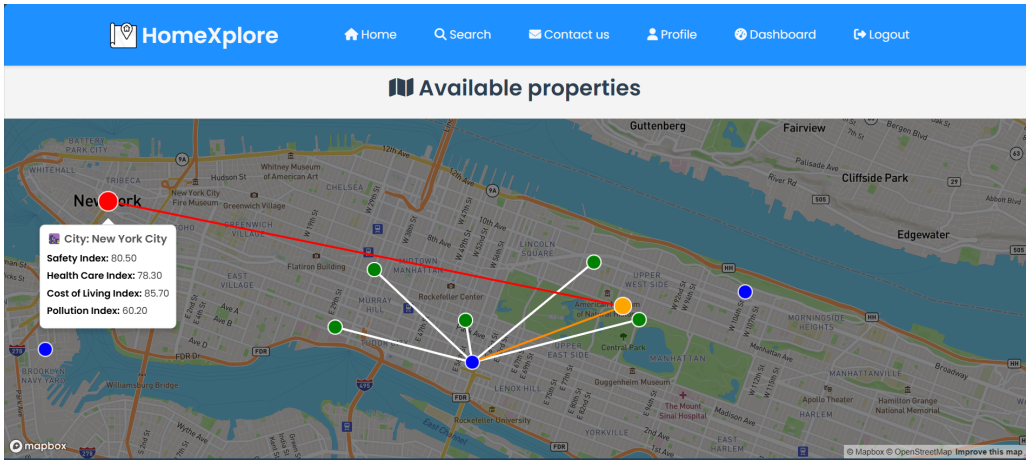

Figure 2.8: Interactive Map Feature (map.html) - Map View 2

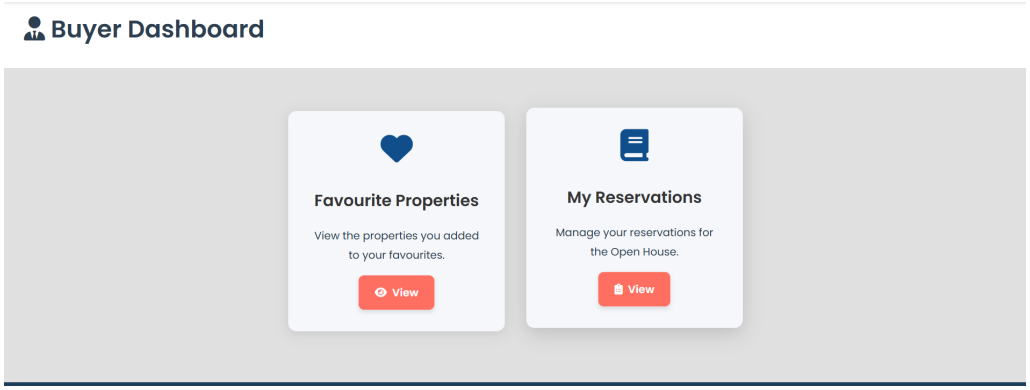

Figure 2.9: Buyer Dashboard (buyer.html)

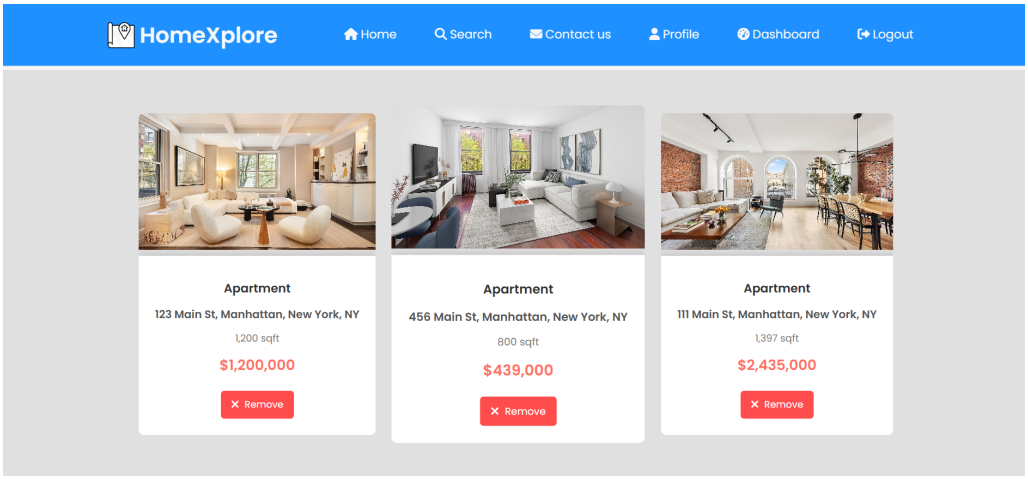

Figure 2.10: Favourite Properties (favourite.html)

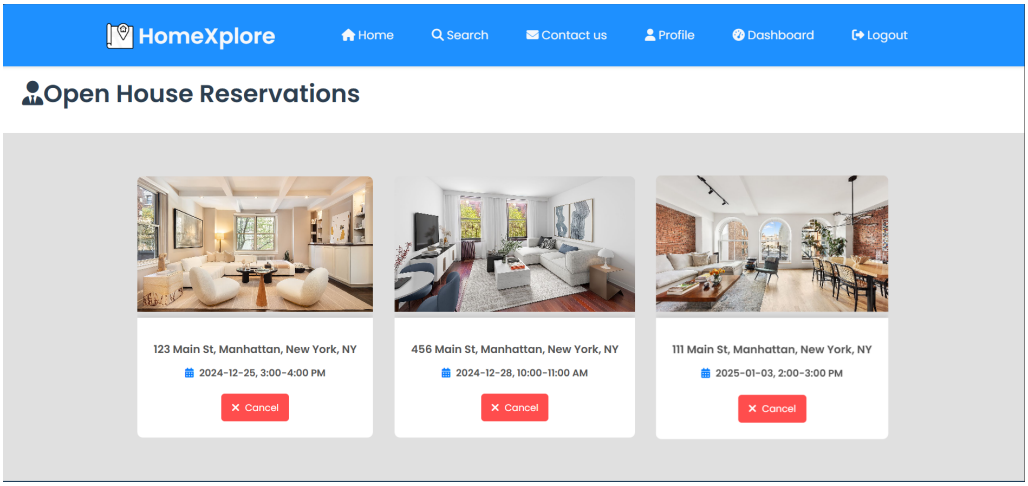

Figure 2.11: My Reservations (buyer_reservations.html)

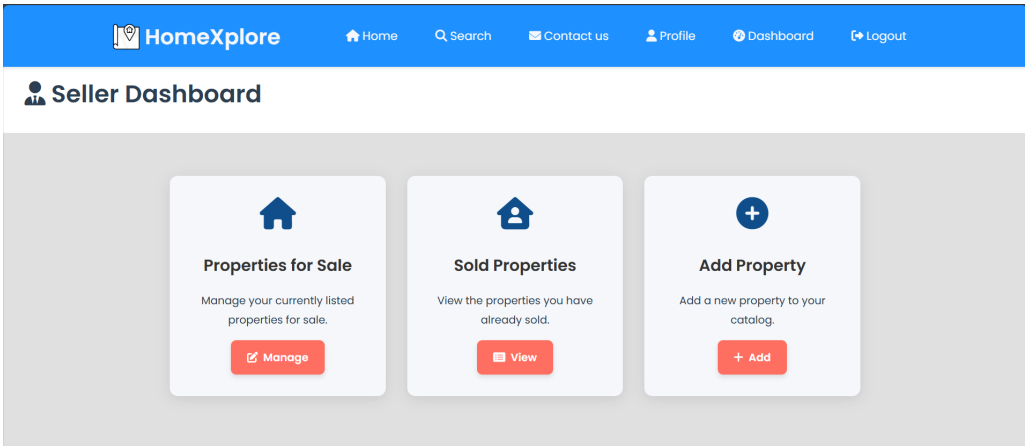

Figure 2.12: Seller Dashboard (seller.html)

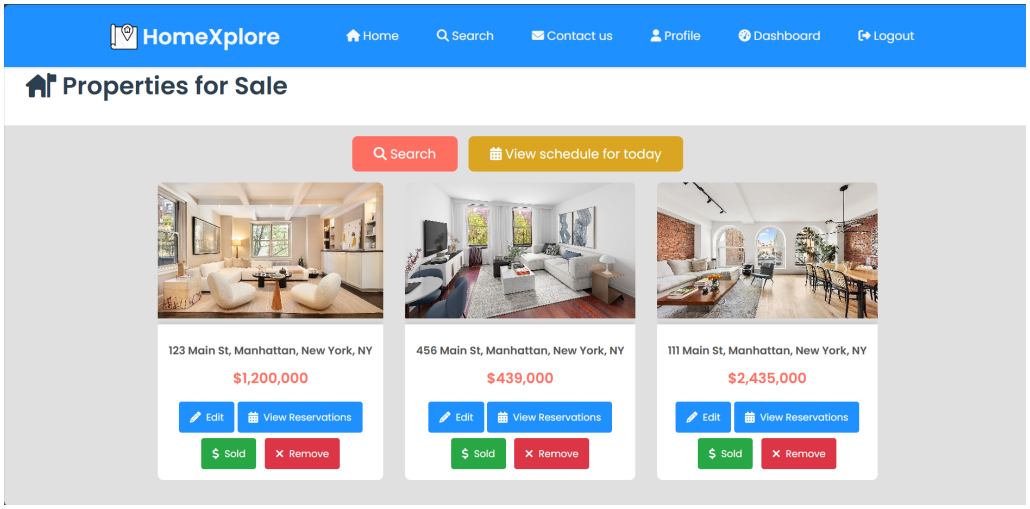

Figure 2.13: Active Listings (seller_for_sale.html)

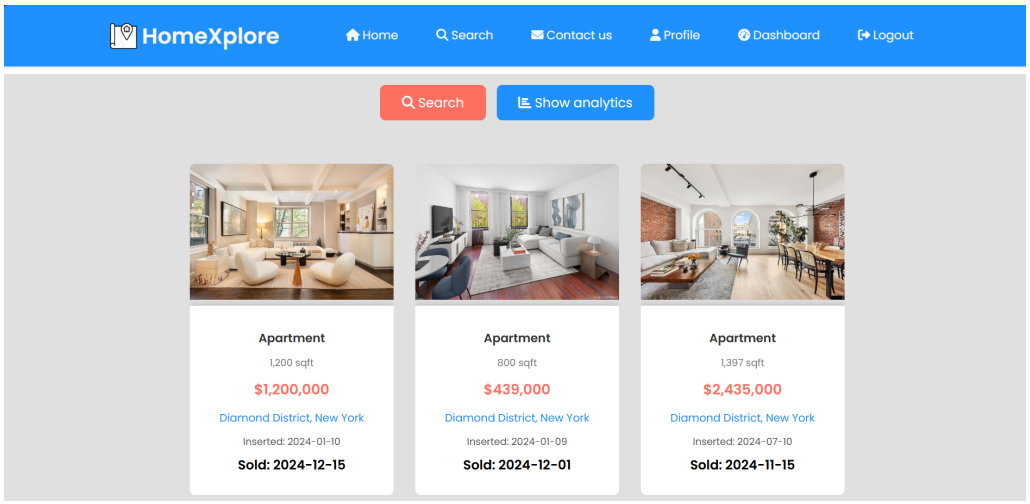

Figure 2.14: Sold Properties (seller_sold.html)

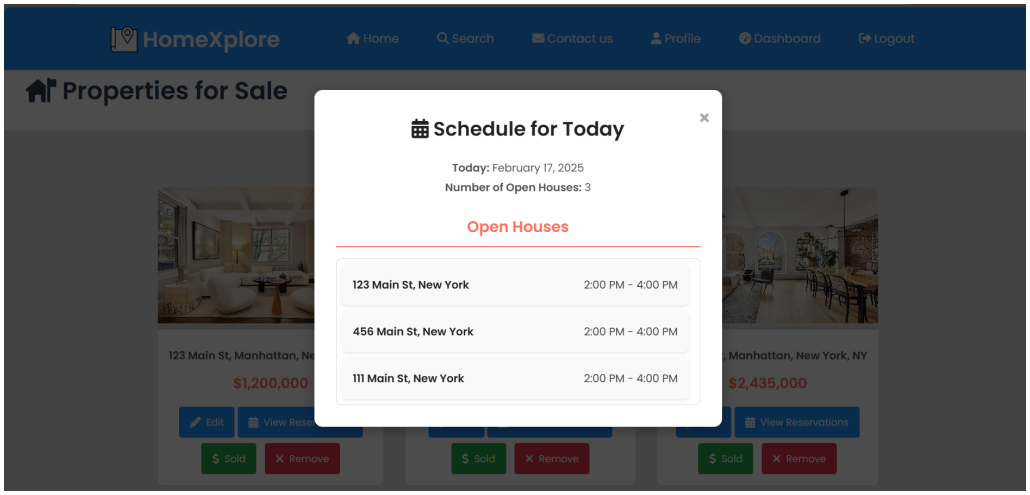

Figure 2.15: Open House Schedule (seller_for_sale.html)

Chapter 3

Design

3.1 Actors

The main actors in *HomeXplore* are:

- **Guest**: a user who explores available properties without registering. A guest can:
  - View all properties for sale and perform filtered searches based on criteria such as price, city, neighbourhood, and area.
  - Use the Geo Exploration feature to understand the surroundings of a property and receive suggestions for similar listings.
  - Register as a buyer or seller to unlock additional platform features.
- **Buyer**: a registered user actively looking to purchase a property. In addition to all guest functionalities, a buyer can:
  - Book appointments for Open House events.
  - Save favorite properties for easy access.
  - Access real estate market statistics.
- **Seller**: a registered user, typically representing a real estate agency, aiming to reach potential buyers through the platform. A seller can:
  - List properties for sale.
  - Organize Open House events and track attendees.
  - View and manage both active and sold property listings.
  - Monitor sales performance through analytics.
  - Access real estate market statistics.

3.2 Requirements

3.2.1 Functional Requirements

Guest

- Browse the platform without registration.
- Register as a buyer or seller.
- View complete details of any property.
- Search for properties matching specific criteria (e.g., address, city, price, area).
- Access the Geo Exploration feature to:
  - View nearby points of interest.
  - Check the liveability score of a property.
  - Learn about the neighbourhood and city where the property is located.
  - Receive recommendations for similar properties.

Buyer

- Log in and log out of the platform.
- Access all guest functionalities (excluding registration).
- Update profile information.
- View a summary of properties saved in the favourites list.
- Manage the favourites list (add or remove properties).
- Register for Open House events.
- View scheduled reservations for upcoming Open House events.
- Cancel an Open House reservation.
- Access real estate market statistics (see details in the Analytics chapter).

Seller

- Log in and log out of the platform.
- Access all guest functionalities (excluding registration).
- Update profile information.
- List properties for sale.

- Modify details of listed properties, including scheduling Open House events, uploading photos, adjusting prices, updating descriptions, and modifying other static information.
- View summary information about active listings.
- Mark a property as sold.
- View summary information about sold properties.
- Monitor sales performance through analytics (see details in the Analytics chapter).
- View the list of reservations for an Open House event.
- View his appointments for the day.
- Access real estate market statistics (see details in the Analytics chapter).

3.2.2 Non-Functional Requirements

- User passwords must be encrypted.
- Operations restricted to certain user types must be properly protected.
- The application must be intuitive and user-friendly.
- The system must be stable, provide consistent results, and handle exceptions appropriately.
- High availability is a priority, even if some data is occasionally outdated.
- The codebase should be maintainable and readable.
- The application must be optimized for speed, ensuring quick responses to queries, especially for filtered searches, point of interest browsing, and property recommendations.

3.3 UML Class Diagram

Figure 3.1 illustrates the UML Class Diagram for *HomeXplore*:

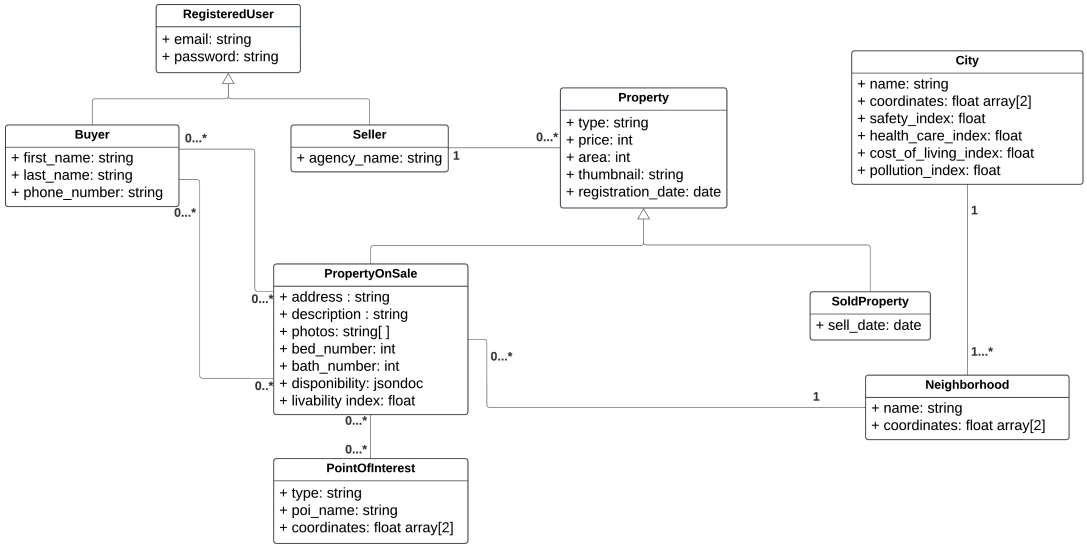

Figure 3.1: UML Class Diagram for HomeXplore

Chapter 4

Data Modelling and Structure

4.1 Dataset

The dataset was built from two main sources: Zillow and OpenStreetMap, complemented by Faker for generating private data and Numbeo for obtaining real data for cities.

Zillow is an online real estate marketplace that allows users to search for properties for sale or rent, view property details and photos, and get estimates of home values.

OpenStreetMap, instead, is a collaborative project that creates a free, editable map of the world by collecting detailed geospatial data.

4.1.1 Zillow

For Zillow, the scraping algorithm `zillow_scraper.py` was designed. Its main features include using appropriate headers to simulate a browser and analyzing the raw HTML content to identify the portion related to the properties. The algorithm also iterates through the result pages to gather all available data. The scraping was performed for the 19 most populated cities in the United States by inputting the city name and the state's acronym.

Zillow Data

The following table summarizes key information for each city scraped from Zillow:

City	State	Population	Density (per km^2)	Dataset Size
New York	New York	8,336,697	26,403.8	3.2 Mb
Los Angeles	California	3,857,799	7,876.4	4.4 Mb
Chicago	Illinois	2,714,856	12,752.2	3.6 Mb
Houston	Texas	2,160,821	3,371.8	4.2 Mb
Phoenix	Arizona	1,660,272	1,235.4	4.5 Mb
Philadelphia	Pennsylvania	1,547,607	11,232.8	3.8 Mb
San Antonio	Texas	1,382,951	2,808.3	4.2 Mb
San Diego	California	1,338,348	3,772.4	4.4 Mb
Dallas	Texas	1,241,162	3,470.3	3.8 Mb
Austin	Texas	974,233	970.9	4.1 Mb
Jacksonville	Florida	971,319	2,162.8	4.6 Mb
San Jose	California	968,237	5,116.9	3.9 Mb
Indianapolis	Indiana	834,852	2,610.6	4.2 Mb
San Francisco	California	825,863	16,632.4	4.1 Mb
Columbus	Ohio	809,798	3,383.1	4.1 Mb
Fort Worth	Texas	777,992	1,828.0	3.9 Mb
Charlotte	North Carolina	775,202	2,232.1	4.1 Mb
Detroit	Michigan	701,475	6,853.5	3.6 Mb
Oklahoma City	Oklahoma	681,054	3,938.8	4.3 Mb

Table 4.1: Zillow data for the 19 most populated U.S. cities.

The scraping results include not only property information (used to create the `PropertyOnSale` collection in MongoDB and the nodes in Neo4j) but also the names of the agencies selling the properties and the schedule (day and time) of the open house events. This data is used to create the `Seller` collection in MongoDB and the `ReservationsSeller` and `ReservationsBuyer` in Redis.

4.1.2 OpenStreetMap

OpenStreetMap was chosen as the source of geospatial data via the Nominatim and Overpass APIs, which provide limited-rate access sufficient to meet the application's requirements. This data was used to create the `City`, `Neighbourhood`, and `POI` nodes in Neo4j.

Cities and Neighbourhoods

Using the script `getCitiesandNeighbourhoods.ipynb`, we obtained the following results for cities and the number of neighbourhoods in each:

City	State	Number of Neighbourhoods
New York	New York	150
Los Angeles	California	51
Chicago	Illinois	30
Houston	Texas	154
Phoenix	Arizona	19
Philadelphia	Pennsylvania	4
San Antonio	Texas	143
San Diego	California	23
Dallas	Texas	24
Austin	Texas	11
Jacksonville	Florida	37
San Jose	California	30
Indianapolis	Indiana	12
San Francisco	California	43
Columbus	Ohio	116
Fort Worth	Texas	3
Charlotte	North Carolina	100
Detroit	Michigan	10
Oklahoma City	Oklahoma	76

Table 4.2: Cities and number of neighbourhoods.

Points of Interest (POIs)

For points of interest, we selected a subset of data for each city using the script `getPOIs.ipynb`. The following types of POIs were included:

- amenity: hospital
- amenity: school
- leisure: park
- amenity: police
- shop: supermarket
- amenity: kindergarten
- industrial: factory
- landuse: landfill
- amenity: prison
- amenity: grave_yard

4.1.3 Faker

The Faker library was used to generate private data (name, surname, email, and password) for the `Buyer` and `Seller` collections in MongoDB. Passwords were hashed using `bcrypt`.

4.1.4 Numbeo

Numbeo, the world's largest cost-of-living database, was used to obtain the following city indexes:

- safety_index
- health_care_index
- cost_of_living_index
- pollution_index

Numbeo acts as a crowd-sourced global database of quality-of-life data.

4.2 Scripts and Functions for Dataset Assembly

Once the data was scraped or gathered from the APIs, most of it required further processing before it could be used by the bulk module; only `cities.csv` and `neighbourhoods.csv` were finalized.

4.2.1 Properties and Sellers

The scraping process generated a CSV file for each city. Using the Jupyter Notebook script `formatScrapedPropertiesIntoMongoDBCollectionsandPropertyNeo4J.ipynb`, the CSVs were analyzed one by one. The agencies were extracted into a `sellers.csv` file, which also included embedded properties on sale and, randomly, sold properties selected from the scraped data.

- The correct neighbourhood was assigned to each property by referencing the `neighbourhoods.csv` file obtained from OpenStreetMap, using a proximity-based approach.
- Based on these relationships, a bulk file for the properties on sale in Neo4j was also created.
- During this process, data cleaning and parsing were performed to fix the format of the open house dates and to prevent data validation errors.

4.3 Buyers

Initially, buyers were generated with Faker within the same script as the properties. However, to allow more flexibility in the amount of generated data, a new file called `formatBuyers.ipynb` was created. This script gathers data from the previous process to assign favourite properties to buyers.

4.4 POIs

The POIs were unified into a single file for Neo4j, `pois.csv`, using the script `formatPOIs.ipynb`.

4.5 Reservations

Reservations were generated using the script `formatReservations.ipynb`, which takes as input the `sellers.csv`, `buyers.csv`, and `properties_on_sale.csv` files from the previous scripts. An auxiliary function was used to obtain the next date based on the current date and the day specified in the disponibility.

4.6 Databases

We have chosen to use **MongoDB**, **Redis**, and **Neo4j** to develop our application in the most efficient, appropriate, and modular way. This allows us to leverage the strengths of each database to enhance the functionality of our system.

4.6.1 Volume Consideration

The table below is used to illustrate the daily number of occurrences for each of the most important and frequent operations in the application. The following volumes are considered for these calculations:

- 50,000 total users interested in buying (10,000 registered buyers, 40,000 guests)
- 25,000 total properties on sale
- 3,000 sellers

The estimated number of active users per day is:

- 8,000 guests (20%)
- 3,000 buyers (33%)
- 2,400 sellers (80%)

The *# of Occurrences* should be interpreted as the weighted average number of daily accesses per user, taking into account the different types of users.

An "operation" on a database is defined as a single call to the database (for example, a `findOne(...)` is considered a single operation, regardless of its complexity).

API	Number of Occurrences	MongoDB, Neo4j, Redis Ops	Total Ops(per day)
Log in	$2,400 + 3,000$	1, 0, 0	5,400
See property page	$4 \times 5 \times 8,000 + 2 \times 10 \times 3,000$	1, 0, 0	220,000
Search properties	$4 \times 8,000 + 2 \times 3,000$	1, 0, 0	38,000
See properties on the map	$4 \times 2.5 \times 8,000 + 2 \times 7 \times 3,000$	0, 1, 0	122,000
See POIs related to a property	$4 \times 2.5 \times 8,000 + 2 \times 7 \times 3,000$	0, 1, 0	122,000
View favourite properties	$0.5 \times 3,000$	1, 0, 0	1,500
Add favourite property	$2 \times 1 \times 3,000$	1, 0, 0	6,000
View reservations (buyer)	$1 \times 3,000$	0, 0, 1	3,000
Add reservation (buyer)	$12,000$	0, 0, 4	48,000
Search properties on sale (seller)	$12,000$	1, 0, 0	12,000
Add property on sale	$0.1 \times 2,400$	2, 4, 0	1,440
Sell property on sale	$0.1 \times 2,400$	3, 1, 1	1,200
View appointments for the day	$1 \times 2,400$	1, 0, 0	2,400
View reservations for a property	$5 \times 1 \times 2,400$	0, 0, 1	12,000

Table 4.3: Summary of Daily Operation Volumes

Some observations about the table

The assumptions behind the numbers in the table are explained here:

- A guest performs **4 searches per session**, and for each search, they are interested in **5 houses**.
- A buyer performs **2 searches per session**, and for each search, they are interested in **10 houses**, as they seek more detailed information compared to guests.
- The estimated ratio of users accessing the map is:
  - **50%** of the houses guests are interested in.
  - **70%** of the houses buyers are interested in.
- Users who view a house on the map are also interested in seeing adjacent points of interest (100% of cases).
- Buyers check their **favourites list once every two accesses**.
- **10% of the houses** viewed by buyers are added to their favourites.
- Buyers check their **appointments once per session** and schedule an appointment **once every 10 sessions**.
- Sellers view the **properties they have for sale at least once every 5 sessions**.
- Sellers **add a new property once every 10 sessions**, the same estimated frequency as property sales.

4.7 MongoDB

MongoDB was chosen to store most of the application's data, particularly information related to houses, users, and their interactions with properties. We selected MongoDB for its ability to efficiently handle complex data structures using JSON-like documents and its flexibility in managing heterogeneous data.

In particular, its capability to store related data within a single document rather than distributing it across multiple tables simplifies queries and improves performance. Additionally, we take advantage of MongoDB's denormalization features by adding redundant data across collections to optimize query speed for users.

Moreover, thanks to its efficient indexing system and powerful aggregation framework, MongoDB enables us to manage large volumes of data effectively.

4.7.1 UML Class Diagram Components

Entities

- PropertyOnSale
- SoldProperty
- Property
- Seller
- Buyer
- RegisteredUser

Relationships

- PropertyOnSale - Buyer
- Seller - Property

Generalizations

- Property
- RegisteredUser

4.7.2 Collections

The MongoDB database is structured into the following collections:

Seller

```
{
  "_id": ObjectId,
  "agency_name": "string",
  "email": "string",
  "password": "string",
  "property_on_sale": [
      {
        "_id": ObjectId,
        "city": "string",
        "neighbourhood": "string",
        "address": "string",
        "price": int32,
        "thumbnail": "string",
        "disponibility": {
            "day": "string",
            "time": "string",
            "max_attendees": int32
        }
      }
  ],
  "sold_property": [
      {
        "_id": ObjectId,
        "city": "string",
        "neighbourhood": "string",
        "price": int32,
        "thumbnail": "string",
        "type": "string",
        "area": int32,
        "registration_date": ISODate,
        "sell_date": ISODate
      }
  ]
}
```

Buyer

```
{
    "_id": ObjectId,
    "name": "string",
    "surname": "string",
    "email": "string",
    "password": "string",
    "phone_number": "string",
    "favourites": [
      {
```

```
        "_id": ObjectId,
        "thumbnail": "string",
        "address": "string",
        "price": int32,
        "area": int32
      }
    ]
}
```

PropertyOnSale

```
{
  "_id": ObjectId,
  "city": "string",
  "neighbourhood": "string",
  "address": "string",
  "price": int32,
  "thumbnail": "string",
  "type": "string",
  "bed_number": int32,
  "bath_number": int32,
  "area": int32,
  "description": "string",
  "photos": ["string"],
  "registration_date": ISODate,
  "disponibility":{
    "day": "string",
    "time": "string",
    "max_attendees": int32
  }
}
```

4.7.3 Motivation

The structure of the collections is driven by the functional requirements of our application, specifically:

Guest

- Register as a buyer or seller.
- View complete details of any property.
- Search for properties based on specific criteria (e.g., address, city, price, area).

Buyer

- View a summary of properties saved in the favourites list.
- Manage the favourites list (add or remove properties).
- Access real estate market statistics (detailed in the analytics chapter).

Seller

- List a property for sale.
- Modify details of a listed property, including scheduling Open House events, uploading photos, changing the price, updating the description, and modifying static information.
- View summary information about listed properties.
- Mark a property as sold.
- View a summary of sold properties.
- Monitor sales performance through analytics (detailed in the analytics chapter).
- Access real estate market statistics (detailed in the analytics chapter).

Generalizations and Entities

The `RegisteredUser` generalization is resolved into two separate entities, `Seller` and `Buyer`, with attributes from the parent entity moved into the child entities. This decision is based on the fact that our application has two main user types, each accessing distinct functionalities. Since there is no need to retrieve information about both user types simultaneously, keeping them separate ensures a clearer and more efficient data structure.

Similarly, the `Property` generalization is resolved by eliminating the parent entity and distributing its attributes and relationships among the child entities. This approach is necessary because these child entities maintain different relationships, and access to them is always independent rather than simultaneous.

The `PropertyOnSale` collection implements the `PropertyOnSale` entity with its generalization already resolved. This ensures that all users can search through all properties for sale and access the necessary information. Additionally, market analytics are based on the entire set of properties currently on sale.

Relationships

The many-to-many relationship between `Buyer` and `PropertyOnSale`, which models the *Favourites List* functionality, is implemented using replication and partial document embedding of `PropertyOnSale` within the `Buyer` collection.

Only the minimal amount of necessary information is replicated to avoid querying the `PropertyOnSale` collection when a buyer views their favourites list. Given that `PropertyOnSale` is the largest collection in terms of document count, implementing a join via document linking would be costly. Since the *View Favourites List* operation is expected to be performed multiple times per session, this approach ensures better efficiency and performance.

The one-to-many relationship between `Seller` and `PropertyOnSale`, which results from the generalization resolution, is implemented through partial document embedding of `PropertyOnSale` within the `Seller` collection.

To minimize replication, only essential property information is duplicated, enabling sellers to query their own listings without accessing the `PropertyOnSale` collection. The most frequent operation for sellers is viewing a summary of their listed properties. This operation is crucial for checking the list of bookings for Open House events and serves as a basis for potential property updates.

The most commonly modified attributes are expected to be price and disponibility, while other details remain largely static, defined at the time of listing creation. This collection structure also optimizes monitoring analytics for sellers, further reducing direct access to the `PropertyOnSale` collection.

The one-to-many relationship between `Seller` and `SoldProperty`, derived from the `Property` generalization resolution, is implemented through document embedding of the `SoldProperty` entity within the `Seller` collection.

This approach was chosen because there is no need for global queries or analytics on all sold properties. Each seller only accesses their own sold properties, and sales monitoring analytics are performed exclusively on a seller's transactions. As access to sold properties is always filtered by the seller, embedding provides an efficient solution without unnecessary queries.

4.8 Redis

Redis is used to manage reservations for open house events because key-value databases are the most suitable for handling large amounts of volatile and fast data such as reservations. It is also optimized for read-intensive applications.

4.8.1 UML Class Diagram Components

Entities

- PropertyOnSale
- Buyer

Relationships

- PropertyOnSale - Buyer

4.8.2 Structure

The Redis structure consists of two sets of keys with the following schema:

Reservations (Seller)

Key: `property_on_sale_id:<property_on_sale_id>:reservations_seller`
Value: A list of JSON objects containing buyer details.

Example:

```
property_on_sale_id:65c4b1f7e8d4a3a1b2c3d4e5:
   reservations_seller
[
  {"buyer_id": "65c4b1f7e8d4a3a1b2c3d4e5", "full_name": "
   John Doe", "email": "john@example.com", "phone": "
   +123456789"},
  {"buyer_id": "65c4b203f9a5b4c6d7e8f9a0", "full_name": "
   Jane Smith", "email": "jane@example.com", "phone": "
   +987654321"}
]
```

Reservations (Buyer)

Key: `buyer_id:<buyer_id>:reservations_buyer`
Value: A list of JSON objects containing details about the open house events.

Example:

```
buyer_id:65c4b1f7e8d4a3a1b2c3d4e5:reservations_buyer
[
  {
    "property_on_sale_id": "65c4b1f7e8d4a3a1b2c3d4e5",
    "date": "2025-02-17",
    "time": "9:00 AM - 11:00 AM",
    "thumbnail": "example_url.com",
    "address": "35 Example St., New York, NY"
  },
  {
    "property_id": "65c4b203f9a5b4c6d7e8f9a0",
    "date": "2025-02-18",
```

```
    "time": "1:00 PM - 2:30 PM",
    "thumbnail": "example2_url.com",
    "address": "77 Example St., New York, NY"
  }
]
```

4.8.3 Motivation

The choice of this key-value database structure is driven by the following functional requirements:

Buyer

- View scheduled reservations for upcoming Open House events.

Seller

- View the list of reservations for an Open House event.

Entities

Through the two sets of keys, we are implementing a many-to-many relationship between `Buyer` and `PropertyOnSale`. The `Buyer` and `PropertyOnSale` entities do not have a direct counterpart in Redis, as they are implemented in MongoDB. In Redis, we replicate only certain attributes of these entities to optimize operations, reducing the need to access MongoDB frequently.

Specifically, for each buyer's reservation on a property, we replicate the property's thumbnail, address, `_id`, and the buyer's contact information. These details are stored in two separate lists of JSON documents, representing the reservation from both the buyer's and seller's perspectives. Additionally, in the buyer-side JSON document, we include the date and time of the next available appointment, based on the `disponibility` attributes of the property stored in MongoDB.

Relationships

The many-to-many relationship is implemented using lists of JSON documents as values for the keys. This design choice is driven by efficiency: read operations have an $\mathcal{O}(1)$ time complexity, making data access extremely fast. By leveraging this characteristic, reservations can be retrieved in a single access operation, ensuring a seamless user experience.

Reservations are always accessed contextually by both buyers and sellers:

- **Buyers:** They need to view their complete list of reservations to keep track of scheduled Open House events, with key details serving as reminders.

- **Sellers:** They need to see the details of all buyers registered for a specific Open House event to manage attendance, contact participants if needed, or address any event-related issues.

Moreover, this structure enables sellers to better organize Open House events based on the number of reservations received. For example:

- If an event has a high number of reservations, the seller can arrange for additional staff.
- If the number of reservations is low, the seller may consider extra marketing strategies to boost interest.

Additionally, sellers have the possibility to export the list of interested buyers before an event. A dedicated button on the seller's property page allows them to generate a PDF with the reservation details, facilitating event management and post-event follow-ups.

With a simple key, a buyer can immediately retrieve the list of their booked Open House events without executing complex joins across multiple tables. Similarly, a seller can obtain the full list of participants for an event with a single read operation.

Reservation Expiration and Cleanup

The structure is designed to efficiently handle the volatility of reservations. Specifically:

- **On the seller's side:** When an Open House event expires, all associated reservations are automatically removed using a Time-To-Live (TTL) mechanism. This ensures that outdated reservations linked to a specific property are deleted as soon as the event has passed.
- **On the buyer's side:** Reservations are cleaned up dynamically. When a buyer checks their upcoming appointments, the system verifies if any past reservations have exceeded their scheduled time. If so, those expired reservations are automatically removed.

This approach ensures that both buyers and sellers have an up-to-date and efficiently managed reservation system, minimizing unnecessary data persistence while maintaining real-time accuracy.

4.9 Neo4j

4.9.1 Overview

We used Neo4j to implement the geo-exploration functionality of our application. The natural graph structure of geographical data such as neighbourhoods, cities, points of interest, properties, and their relationships makes Neo4j the ideal choice

for modelling this type of data. Additionally, Neo4j provides numerous optimized geospatial queries that significantly reduce the cost of inserting a new property into the graph.

4.9.2 UML Class Diagram Components

Entities

- City
- PropertyOnSale
- Neighbourhood
- PointOfInterest

Relationships

- PropertyOnsale - PointOfInterest
- PropertyOnSale - Neighbourhood
- Neighbourhood - City

4.9.3 Structure

Nodes and Properties

```
City (NODE)
    Label: City
    Properties:
      - name: String - Name of the city
      - coordinates: POINT - Latitude and longitude
      - safety_index: FLOAT
      - health_care_index: FLOAT
      - cost_of_living_index: FLOAT
      - pollution_index: FLOAT

Neighbourhood (SUPER NODE)
    Label: Neighbourhood
    Properties:
      - name: String - Name of the neighbourhood
      - coordinates: POINT - Latitude and longitude

POI (NODE)
    Label: POI
    Properties:
      - name: String - Name of the POI
      - coordinates: POINT - Latitude and longitude
```

```
      - type: String - Type of POI (e.g., school, park,
   hospital)

Property (NODE)
    Label: Property
    Properties:
      - coordinates: POINT - Latitude and longitude
      - price: FLOAT - Price of the property
      - thumbnail: String
      - type: String
      - score: FLOAT - Score calculated based on
   proximity to POIs
      - _id: MongoDBIndex - Index inside MongoDB
```

Relationships and Relationship Types

```
NEIGHBOURHOOD_BELONGS_TO_CITY
    Type: BELONGS_TO_CITY
    From: Neighbourhood
    To: City

PROPERTY_LOCATED_IN_NEIGHBOURHOOD
    Type: LOCATED_IN_NEIGHBOURHOOD
    From: Property
    To: Neighbourhood

PROPERTY_NEAR_POI
    Type: NEAR
    From: Property
    To: POI
    Properties:
      - distance: FLOAT - Distance in meters between the
   property and the POI

PROPERTY_NEAR_PROPERTY
    Type: NEAR
    From: Property
    To: Property
    Properties:
      - distance: FLOAT - Distance in meters between the
   property and another property
```

4.9.4 Motivation

The structure of the graph is driven by the functional requirements of our application, specifically:

- View nearby points of interest.
- Check the liveability score of a property.
- Learn about the neighbourhood and city where a house is located.
- Receive recommendations for similar properties.

4.9.5 Entities

The UML diagram entities `City`, `Neighbourhood`, and `PointOfInterest` are implemented in Neo4j as nodes with the same names, each containing all the attributes defined in the respective entities. The `PropertyOnSale` node, on the other hand, replicates in the graph database the minimal information already present in the document database regarding properties on sale, in order to optimize graph operations. Additionally, it contains the `_id` of the property document from the MongoDB `PropertyOnSale` collection. This allows all information about a specific property to be concurrently retrieved, should the user wish to obtain further details while exploring the map.

4.9.6 Relationships

The many-to-many relationship between `PropertyOnSale` and `PointOfInterest` is implemented via the unidirectional `PROPERTY_NEAR_POI` relationship, which goes from the `PropertyOnSale` node to the `POI` node. This edge is unidirectional because graph operations only require traversal from a property to its connected points of interest, particularly to satisfy the functional requirement *"View nearby points of interest"*. Moreover, creating a unidirectional edge helps to keep the graph lightweight.

The one-to-many relationship between `PropertyOnSale` and `Neighbourhood` is implemented using the unidirectional `PROPERTY_LOCATED_IN_NEIGHBOURHOOD` relationship, from the `PropertyOnSale` node to the `Neighbourhood` node. This edge is unidirectional for the same reasons mentioned above, applied to the functional requirement *"Learn about the neighbourhood and city where a house is located"*.

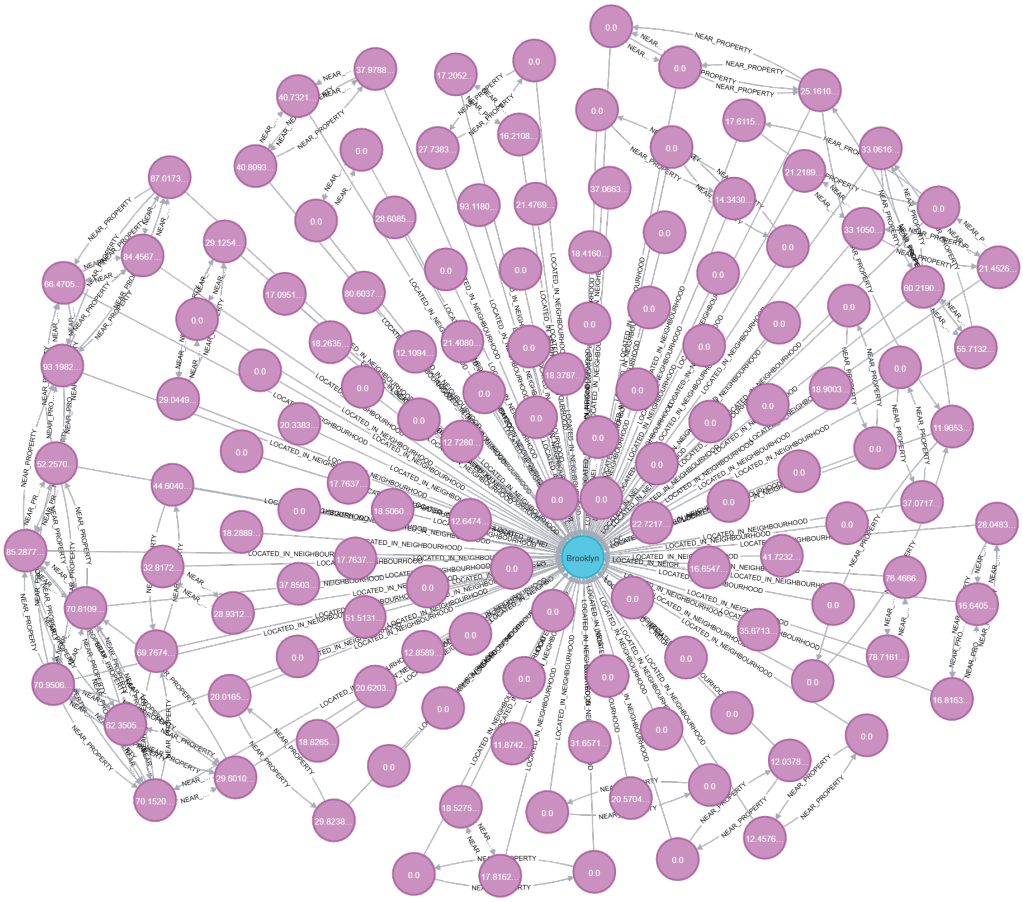

Figure 4.1: Properties in a Neighbourhood

The one-to-many relationship between `Neighbourhood` and `City` follows the same implementation logic as the previous relationships, also based on the functional requirement *"Learn about the neighbourhood and city where a house is located"*.

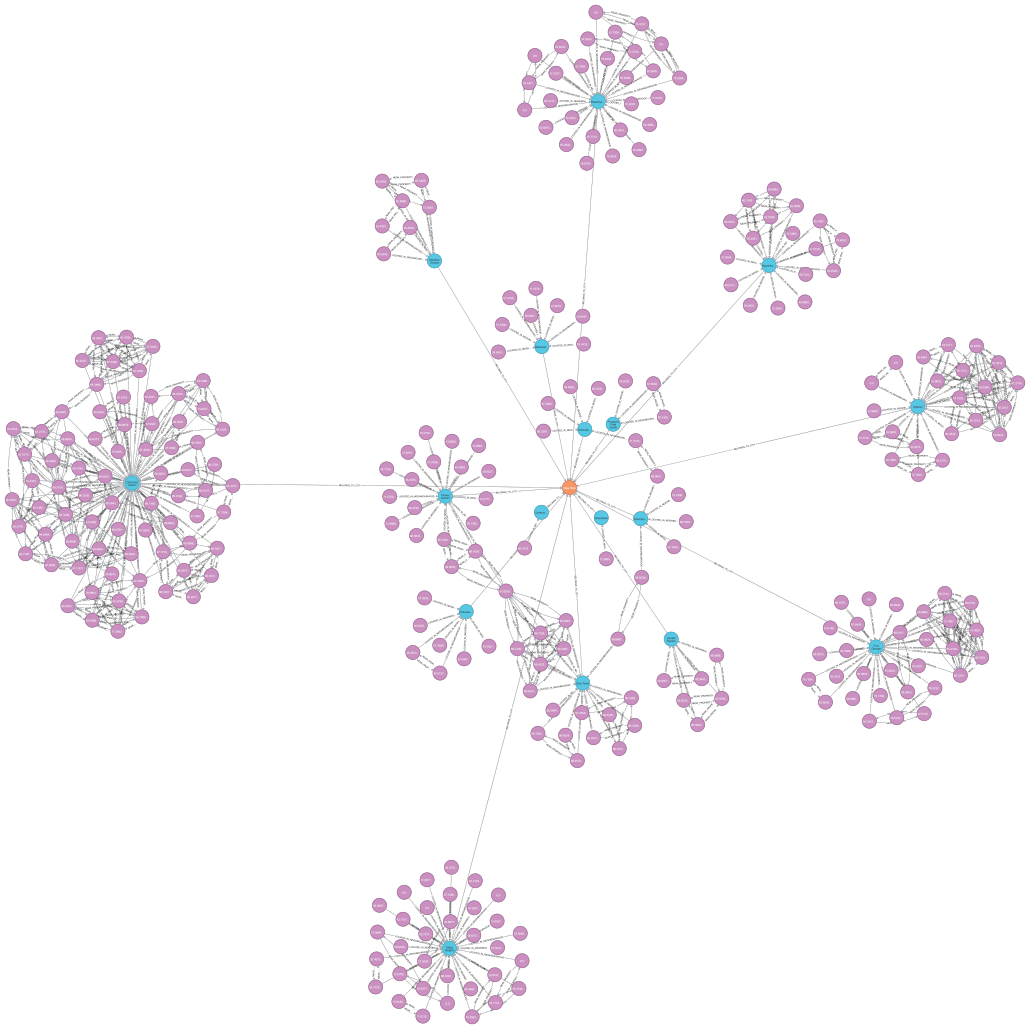

Figure 4.2: Properties and Neighbourhood in a City

The edge between two `Property` nodes is bidirectional, as it must be possible to retrieve suggestions for nearby houses starting from any property in the application. Therefore, each property must be connected to all properties that represent its "neighbours."

Other Motivations

We decided to use Neo4j to implement the geo-exploration functionality instead of MongoDB because the operations required by this functionality are better suited and more efficient on a graph structure than on a document database. Furthermore, the costs associated with data replication are minimal. In Neo4j, the entire MongoDB `PropertyOnSale` collection is replicated with only four attributes for each property (`_id`, `type`, `price`, `thumbnail`). Once the graph is constructed, all functionalities in Neo4j are achieved by identifying a node and traversing its related nodes within a maximum of 2 hops.

In MongoDB, the same operations would require the implementation of an index to perform geospatial queries, which have a complexity of $\mathcal{O}(\log n)$, where n is the total number of properties on sale or the number of points of interest. Thanks to our design, this cost is incurred only during the addition of a new property to the graph, when a seller creates a new property. During this operation, the distances between existing nodes and the new node are evaluated once, and edges are created only if these distances fall below a certain threshold. Thereafter, these edges are used to evaluate distances instead of repeatedly executing costly geospatial queries.

Chapter 5

Implementation

5.1 Project Structure

5.1.1 FastAPI

The API framework used in our application is FastAPI, which is built on top of `Starlette` for handling requests and `Pydantic` for data validation. This framework delivers high performance and automatically generates API documentation that adheres to the `OpenAPI` standard, complete with a `Swagger UI` for testing and interaction. Additionally, Pydantic's validation capabilities are leveraged to ensure that every incoming data payload is verified against the predefined backend schemas. When the data does not meet the specified criteria, Pydantic promptly triggers error responses at the corresponding routes. This automatic error enforcement not only maintains the integrity of the API endpoints but also prevents the codebase from becoming cluttered with manual error-handling routines, ultimately leading to cleaner and more maintainable code.

5.1.2 Docker

Docker and Docker Compose are valuable tools in the development process. They allow creating consistent, isolated environments for applications. Docker packages the application and all its dependencies into containers, so it runs reliably everywhere. Docker Compose then helps manage multiple containers at once. This was especially important for a multi-database setup, which uses MongoDB, Neo4j, and Redis, each in its own container. Using Docker Compose made it much easier to configure and manage these interconnected services, speeding up setup and making our testing much more reliable.

During development, two configurations were used:

- A configuration with a single database instance per type.

- A configuration reflecting the final infrastructure topology, as defined in `docker-compose-distrib`

5.1.3 Modules Folder

Inside this folder, all REST controllers are organized by user type, handling HTTP requests to execute the application's functionalities. Within each user-specific folder, there is a file named `user_type_router.py` that contains all controllers managing the operations a user can request, as well as a *models* subfolder. This subfolder includes two files:

- **user_type_models.py**: Defines the classes for the input types of the controllers. This ensures greater control and enables better modelling of user requests by structuring request data as objects. The file also includes class validators to verify input data.
- **response_models.py**: Contains examples of HTTP responses that the controllers can return. These examples appear in the Swagger UI to assist in interpreting the server's responses.

Controllers do not interact directly with the database; instead, they invoke methods from classes defined in the *entities* folder. These methods always return an HTTP code based on the outcome of the database operation they attempted to perform. The controller then interprets and handles these HTTP codes appropriately, raising exceptions when errors occur. Some common error codes handled in this way are:

- **400**: Error in the controller inputs.
- **404**: Data not found in the database.
- **500**: Internal error during database interaction.

In **Swagger**, we provide a brief description of all interfaces, including the required inputs and examples of possible responses. Moreover, in the *Relevant Operation* section, there is a detailed description of all interfaces responsible for handling important operations.

Auth Folder

Within the *modules* folder, there is also an Auth Folder that implements its own logic. Inside this folder, two files are present:

- `JwtHandler.py`: Defines the *JWTHandler* class, which manages all authentication-related operations.
- `auth_helpers.py`: Contains two helper methods that support functions for encoding and decoding passwords.

Authentication and protection of functionalities for registered users work as follows:

- To log in, the buyer or seller must specify the user type, email, and password.
- A call is made to the MongoDB database to verify the entered data (the password is stored in an encrypted form in the database).
- If authentication is successful, a temporary token is created by encrypting the user's `_id` and type. This token is required to access all functionalities that require a registered user (such as viewing analytics or making a reservation).
- Within each route for registered users, the first function called is always `verifyAccessToken`, which extracts the `_id` and user type from the token.

5.1.4 Entities Folder

Inside this folder, the entities from the UML diagram are implemented and organized by the type of database to which they belong. Each entity is modelled using two files:

- **name_entity.py**: Defines the class that models the type of data inserted into a specific database.
- **db_name_entity.py**: Defines the class responsible for interacting with the database.

All interactions with the database involving data defined in *name_ entity.py* occur through methods in *db_ name_ entity.py*. This class includes attributes such as an embedded *name_ entity.py* object and other auxiliary attributes to store query results. The embedded object is used to capture both the input parameters and the output results, depending on the operation performed. This design enhances modularity at both the data type and database levels and aligns with the requirement to structure the code in an object-oriented manner similar to Java.

MongoDB Folder

Buyer

buyer.py defines the *Buyer* class, modeling a JSON document in the `Buyer` collection.

db_ buyer.py defines the class for interacting with MongoDB regarding the `Buyer` collection, containing a single *Buyer* object. Its methods include:

- **get_profile_info(self)**:
  Retrieves the buyer's profile information using the buyer_id and stores it in the embedded buyer object.
- **get_buyer_by_email(self, email: str)**:
  Finds a buyer in the database using their email and stores personal information in the embedded buyer object.

- **create_buyer(self)**: Creates a new buyer in the database with specified personal information and initializes an empty list.
- **update_buyer(self, buyer: Buyer)**:
  Updates the buyer's personal information in the database.
- **delete_buyer_by_id(self, buyer_id: str)**:
  Deletes a buyer from the database using their `_id`.
- **get_favourites(self)**:
  Retrieves the list of favourite properties for a specific buyer.
- **add_favourite(self, buyer_id: str, favourite: FavouriteProperty)**:
  Adds a property to the buyer's favourites list.
- **update_favourite(self, buyer_id: str, property_on_sale_id: str, updated_data: dict)**:
  Updates the details of a favourite property for a specific buyer.
- **delete_favourite(self, buyer_id: str, property_on_sale_id: str)**:
  Removes a property from the buyer's favourites list.

PropertyOnSale

property_on_sale.py defines the *PropertyOnSale* class, modeling a JSON document in the `PropertyOnSale` collection.

db_property_on_sale.py defines the class for interacting with MongoDB regarding the `PropertyOnSale` collection. It contains a *PropertyOnSale* object and four auxiliary attributes for returning query results. Its methods include:

- **filtered_search(self, input: FilteredSearchInput, page: int = 1, page_size: int = 10)**:
  Searches for properties based on multiple user-selected filters.
- **get_6_random_properties(self)**:
  Retrieves 6 random properties from the database.
- **delete_property_on_sale_by_id(self, property_on_sale_id: str)**:
  Deletes a property using its `_id`.
- **create_property_on_sale(self)**:
  Creates a new property listing with the provided details.
- **update_property_on_sale(self)**:
  Updates an existing property listing, including photos, open house availability, and individual property details (e.g., price, city).
- **get_property_on_sale_by_id(self, property_on_sale_id: str)**:
  Retrieves details of a specific property using its `_id`.

- **delete_and_return_property(self, property_on_sale_id: str)**:
  Deletes a property by `_id` and returns its details before deletion.
- **insert_property(self)**:
  Inserts a property into the database with a predefined unique `_id`.
- **get_avg_price_per_square_meter(self, input: Analytics1Input)**:
  Executes analytics 1 (for further details, see reference).
- **get_avg_price_per_square_meter_by_city(self, city: str)**:
  Executes analytics 4 (for further details, see reference).
- **get_statistics_by_city_and_neighbourhood(self, city: str, neighbourhood: str)**:
  Executes analytics 5 (for further details, see reference).

Seller

seller.py defines the *Seller* class, modeling a JSON document in the `Seller` collection.

db_seller.py defines the class for interacting with MongoDB regarding the `Seller` collection. It contains a *Seller* object and three auxiliary attributes for returning query results. Its methods include:

- **create_seller(self)**:
  Creates a new seller in the database by initializing the personal data.
- **get_profile_info(self)**:
  Retrieves the seller's profile information, excluding properties that are currently on sale or already sold.
- **get_seller_by_id(self)**:
  Retrieves a seller from the database using their ID.
- **get_seller_by_email(self, email: str)**:
  Retrieves a seller based on the provided email; this function is used for authentication, which always takes place via email and password.
- **update_seller(self, seller: Seller)**:
  Updates the profile of an existing seller.
- **delete_seller_by_id(self)**:
  Deletes a seller from the database using their ID.
- **get_properties_on_sale(self)**:
  Retrieves the properties currently on sale by a seller.
- **get_property_on_sale_filtered(self, city: str, neighbourhood: str, address: str)**:
  Filters properties on sale based on city, neighbourhood, and address. This

function is useful for the seller to quickly find the properties on sale they are interested in viewing.

- **get_sold_properties_filtered(self, city: str, neighbourhood: str)**:
  Filters sold properties based on city and neighbourhood. This function is useful for the seller to quickly find the sold properties they are interested in viewing.
- **insert_property_on_sale(self, property_on_sale : SellerPropertyOnSale)**:
  Adds a new property to the seller's list of properties on sale.
- **update_property_on_sale(self, property_on_sale : SellerPropertyOnSale)**:
  Updates the details of a property on sale.
- **delete_embedded(self, property_on_sale_id : str)**:
  Removes a property from the seller's list of properties on sale.
- **sell_property(self, property: SoldProperty)**:
  Moves a property from the seller's list of properties on sale to the list of sold properties. This occurs when the seller marks a property as sold.
- **check_property_on_sale(self, property_on_sale_id: str)**:
  Checks whether a property on sale belongs to a specific seller, ensuring that a seller can only delete properties they have actually listed.
- **get_open_house_today(self)**:
  Returns a complete schedule of all open house events scheduled for the current day.
- **get_sold_properties_statistics(self, input: Analytics2Input)**:
  Executes analytics 2 (for further details, see reference).
- **get_avg_time_to_sell(self, input: Analytics3Input)**:
  Executes analytics 3 (for further details, see reference).

Redis Folder

ReservationsBuyer

reservations_buyer.py defines the *ReservationsBuyer* class, which models a value for a single key.

db_reservations_buyer.py defines the class for interacting with Redis regarding *ReservationsBuyer*. It contains a `ReservationsBuyer` object. Its methods include:

- **create_reservation_buyer(self)**:
  Creates a new reservation on the buyer side for the specific buyer who requested the operation. It creates a JSON document containing the necessary details for

the buyer and inserts it into the list of reservations representing the value in the ReservationsBuyer bucket.

- **get_reservation_by_user(self)**:
  Returns the complete list of reservations present in the database associated with a specific buyer.
- **update_reservation_buyer(self)**:
  Updates the data of a specific reservation. This method is useful if the seller changes details related to an Open House event (a rare occurrence).
- **delete_reservations_buyer(self)**:
  Deletes all reservations for a certain buyer. This method is useful if the buyer deletes their account from the platform.
- **delete_reservation_by_property_on_sale_id(self)**:
  Deletes a single reservation from the ReservationsBuyer bucket. In other words, it removes a document from the list of all reservations by modifying the key's value (using the `property_on_sale_id` provided as input).
- **update_expired_reservations(self)**:
  Updates the buyer's reservations list by removing expired reservations and returns the updated list. The update is executed only if there is at least one expired reservation.

ReservationsSeller

db_reservations_seller.py defines the class for interacting with Redis regarding *ReservationsSeller*. It contains a `ReservationsSeller` object. Its methods include:

- **create_reservation_seller(self, day: str, time: str, buyer_id: str, max_attendees: int)**:
  Creates a new reservation associated with a buyer, verifying that the maximum number of attendees has not been exceeded and that the buyer does not already have an active reservation.
- **get_reservation_seller(self)**:
  Retrieves the reservation data for a specific property.
- **update_reservation_seller(self, buyer_id: str, updated_data: dict)**:
  Modifies the details of an existing reservation for a specific buyer. This function is called when a buyer updates their contact information while registered for reservations (a rare event).
- **delete_reservation_seller_by_buyer_id(self, buyer_id: str)**:
  Deletes the reservation of a specific buyer from the reservations list of a property. This is invoked when a buyer cancels a reservation.

- **delete_entire_reservation_seller(self)**:
  Completely removes all reservations associated with a property.
- **update_day_and_time(self, day: str, time: str)**:
  Adjusts the TTL of the value associated with a specific property (i.e., it modifies the moment when the list of contacts for an Open House event will be deleted).
- **handle_book_now_transaction(self, reservation: ReservationS, day: str, time: str, buyer_id: str, max_attendees: int)**:
  Uses a Redis transaction to prevent concurrency conflicts when multiple buyers attempt to book a reservation simultaneously, causing multiple Redis operations to modify the same list. The transaction is necessary to maintain consistency of the list, ensuring that the reservations received by the seller are reliable.

Neo4j Folder

The folders for cities, POI, and neighbourhood contain classes with the same names along with corresponding `db_entity_name.py` files for interacting with these entities, which are implemented as nodes in Neo4j. The methods for interacting with these entities are exclusively CRUD operations. Although these methods are never called directly by the routes, they have been implemented for completeness for all entities stored by the application.

For the create, get, and delete operations, we assume that the neighbourhoods, cities, and points of interest are initially loaded into the database via a loading script and remain unchanged over time. Furthermore, none of our main actors interact directly with these entities; instead, they all interact through `PropertyOnSale`. Consequently, the `PropertyOnSale` entity includes methods to interact with these entities, particularly `GET` methods to retrieve related information.

PropertyOnSaleNeo4j

property_on_sale_neo4j.py defines the `PropertyOnSaleNeo4J` class, which models the *PropertyOnSale* node in Neo4j.
db_property_on_sale_neo4j.py defines the class responsible for handling operations for users interacting with Neo4j. The class, `PropertyOnSaleNeo4JDB`, has as attributes an object of type `PropertyOnSaleNeo4J` and four auxiliary attributes to store query results. Its methods include:

- **get_property_on_sale_neo4j(self)**:
  Retrieves a property on sale from the database using the identifier property_on_sale_id.
- **create_property_on_sale_neo4j(self, neighbourhood_name: str)**:
  Creates a new property on sale in Neo4j, linking it to the neighbourhood where it is located, to nearby POIs, and to nearby properties by establishing

relationships between nodes as described in the structure chapter. “Nearby” refers to all properties and POIs within a 500-meter radius of the property in question. This value may be adjusted; we have chosen to keep it low to speed up the loading time of all properties in the database (approximately 20ms). This operation is the most expensive in Neo4j and is executed within a transaction to ensure strict consistency of data regarding properties on sale in a single Neo4j database. The complexity of the transaction is offset by the efficiency gained in subsequent graph operations, such as viewing nearby points of interest and nearby properties.

- **update_property_on_sale_neo4j(self, update_data: PropertyOnSaleNeo4J, neighbourhood_name: str = None)**:
  Updates the details of an existing property on sale in Neo4j. If provided, it also updates the geographical coordinates and consequently all relationships with other nodes, since the node must be relocated (this operation is very rare, as it presupposes that the seller changes the address of a house, which by definition should never happen).

- **delete_property_on_sale_neo4j(self)**:
  Deletes a PropertyOnSale node from Neo4j along with its relationships.

- **get_city_and_neighbourhood(self)**:
  Retrieves the city and neighbourhood associated with a property on sale by traversing two hops in the graph from a specific property node.

- **get_near_POIs(self)**:
  Retrieves all POIs connected via the NEAR relationship to a specific property.

- **get_near_properties(self)**:
  Retrieves properties near a given property on sale. "Nearby" properties are defined as those directly connected via the NEAR relationship to the property of interest (referred to as "level 1" in the code), and then all properties connected via the NEAR relationship to those "level 1" properties.

- **update_livability_score(self)**:
  Calculates and updates the livability score for a property on sale based on nearby POIs (those connected via the NEAR relationship with the property) and updates this score in the PropertyOnSale node in Neo4j. The formula for calculating the livability score is provided in the appendix.

5.1.5 Bulk Folder

The bulk routes were created to facilitate rapid and efficient development. There are three distinct groups of routes, each corresponding to the population of a specific database. With a simple click in the Swagger UI, it is possible to load all the data gathered from the final CSV files, delete all data present in a database, or

simply verify whether the data has been correctly uploaded. These routes are also crucial because they handle the conversion of CSV properties into the correct formats accepted by the databases (e.g., stringified JSON), ensuring smooth data ingestion and consistency across our systems.

5.1.6 Config Folder

This folder contains the file `config.py`, which loads all configuration settings from a `.env` file, ensuring that sensitive information is managed securely through environment variables.

Chapter 6

Relevant Operations

In this chapter, we describe the most frequent and computationally significant operations of our application. The operations are organized by the database used and by the user type they target.

6.1 MongoDB

6.1.1 Buyer

View Favourites List

This operation displays a buyer's list of favourite properties, presenting summary information for each property to enhance the user experience, in particular thumbnail, price, area and address. It is also shown in the result the _id of the property (even though the buyer won't see it) because it is necessary for the operation *see property page* that extends the current one. It is managed by the REST controller:

```
@buyer_router.get("/favourites", ...)
```

and involves the methods `verifyAccessToken` and `get_favourite`.

Figure 6.1: Get Favourites Controller

After authenticating via `verifyAccessToken`, the controller delegates the database interaction to the *BuyerDB* class. It connects to MongoDB using the `get_favourite` method, which executes a query on the buyer's collection to extract the list of embedded documents, each representing a favourite property. The extracted data is then mapped to the class attribute `favourites`, and an appropriate HTTP status code is returned.

Any errors or data inconsistencies are handled by raising exceptions, which immediately terminate the operation.

6.1.2 Seller

View Properties for Sale

This operation allows a seller to view all his properties currently for sale, along with summary information for each property. The summary includes details that might change over time (such as open house information) as well as static identifiers like the city, neighbourhood, address, thumbnail image, and price.

It is managed by the REST controller:

```
@seller_router.get("/properties_on_sale", ...)
```

which uses `verifyAccessToken` and *get_properties_on_sale*.

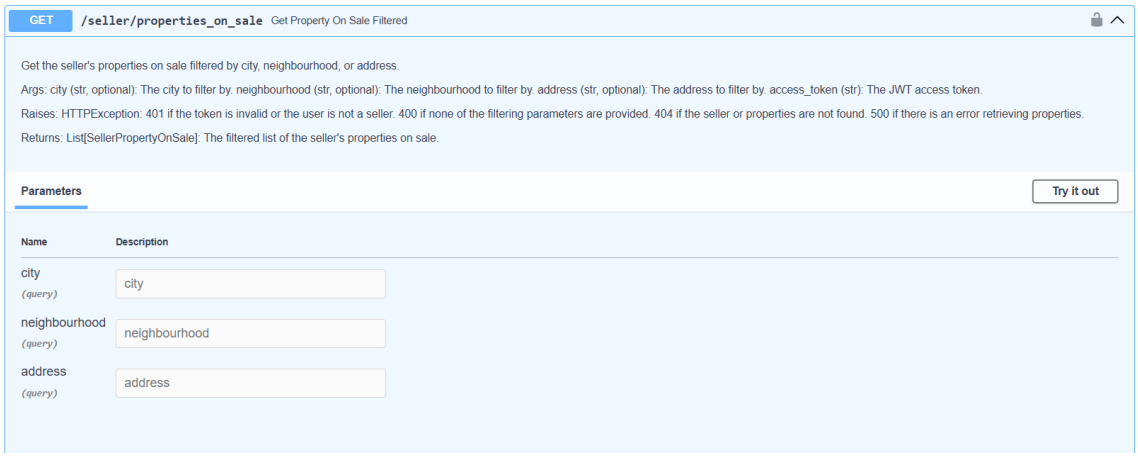

Figure 6.2: Get Property On Sale Controller

After authentication, the controller passes the request to the *SellerDB* class. The `get_properties_on_sale` method executes a query on the seller's collection, extracting a list of embedded documents (each representing a property for sale) from the seller's document. It is also possible to filter by city, neighbourhood and address. The data is then mapped to the class attribute `properties_on_sale`, with the HTTP status code reflecting the query outcome.

Any issues during input validation or query execution are handled by raising exceptions.

```python
def get_property_on_sale_filtered(self, city: str, neighbourhood: str, address: str) -> int:
    mongo_client = get_default_mongo_db()
    if mongo_client is None:
        logger.error("Mongo client not initialized.")
        return 500
    pipeline = [
        { "$match": { "_id": ObjectId(self.seller.seller_id) } },
        { "$project": {
            "_id": 0,
            "properties_on_sale": {
                "$filter": {
                    "input": "$properties_on_sale",
                    "as": "property",
                    "cond": {
                        "$and": [
                            { "$eq": ["$$property.city", city] } if city else {},
                            { "$eq": ["$$property.neighbourhood", neighbourhood] } if neighbourhood else {},
                            { "$eq": ["$$property.address", address] } if address else {}
                        ]
                    }
                }
            }
        }}
    ]
    try:
        result = mongo_client.Seller.aggregate(pipeline)
    except Exception as e:
        logger.error(f"Error retrieving filtered properties for seller {self.seller.seller_id}: {e}")
        return 500
    # Extract the properties_on_sale array from the result
    properties = list(result)[0].get("properties_on_sale", [])
    if not properties:
        return 404
    for property_on_sale in properties:
        property_on_sale["property_on_sale_id"] = str(property_on_sale.pop("_id"))
    self.seller.properties_on_sale = [SellerPropertyOnSale(**property_on_sale) for property_on_sale in properties]
    return 200
```

Create a New Property for Sale (also Neo4j)

This operation allows a seller to list a new property for sale. It is managed by the REST controller:

```
@seller_router.post("/property_on_sale", ...)
```

which first handles authentication and then calls methods to interact with both Neo4j and MongoDB, since the *PropertyOnSale* entity is stored in both databases.

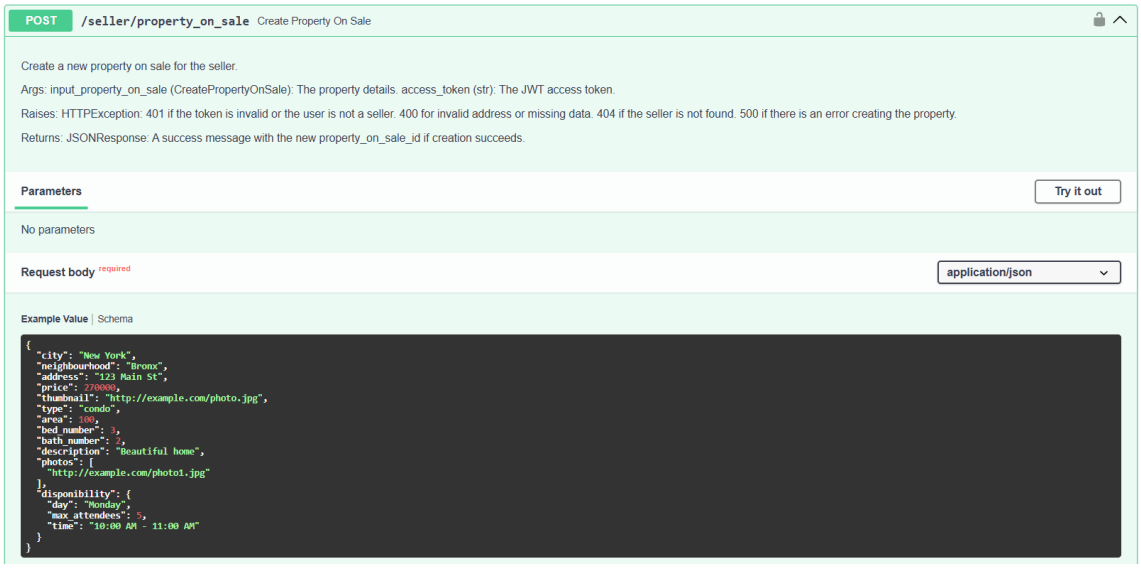

Figure 6.3: Create Property On Sale Controller

```python
@seller_router.post("/property_on_sale", response_model=ResponseModels.CreatePropertyOnSaleResponseModel, responses=ResponseModels.CreatePropertyOnSaleResponses)
def create_property_on_sale(input_property_on_sale: CreatePropertyOnSale, access_token: str = Depends(JWTHandler())):
    seller_id, user_type = JWTHandler.verifyAccessToken(access_token)
    if seller_id is None or user_type != "seller":
        raise HTTPException(status_code=401, detail="Invalid access token")


    geolocator = Nominatim(user_agent="homexplore")
    address = input_property_on_sale.address
    try:
        location = geolocator.geocode(address)
    except Exception as e:
            raise HTTPException(status_code=400, detail="Wrong address.")
    if location is None:
        raise HTTPException(status_code=400, detail="Wrong address.")

    # Insert the property on the property_on_sale collection
    property_on_sale = PropertyOnSale(**input_property_on_sale.model_dump())
    db_property_on_sale = PropertyOnSaleDB(property_on_sale)
    response = db_property_on_sale.create_property_on_sale()
    if response == 400:
        raise HTTPException(status_code=response, detail="Data required.")
    if response == 500:
        raise HTTPException(status_code=response, detail="Failed to create property.")

    # Insert the property on the seller collection
    seller= Seller(seller_id=seller_id)
    seller_db= SellerDB(seller)
    embedded_property_on_sale = SellerPropertyOnSale(**input_property_on_sale.model_dump(exclude={"type", "area", "bed_number", "bath_number", "description", "photos"}))
    embedded_property_on_sale.property_on_sale_id=db_property_on_sale.property_on_sale.property_on_sale_id
    response=seller_db.insert_property_on_sale(embedded_property_on_sale)
    if response != 200:
        if response == 500:
            detail="Failed to create property."
        else:
            detail="Seller not found."
        # Rollback
        response=db_property_on_sale.delete_property_on_sale_by_id(db_property_on_sale.property_on_sale.property_on_sale_id)
        raise HTTPException(status_code=500, detail=detail)
```

```python
#Neo4j

property_on_sale_neo4j = PropertyOnSaleNeo4J(
    property_on_sale_id=db_property_on_sale.property_on_sale.property_on_sale_id,
    **input_property_on_sale.model_dump(include={"price", "type", "thumbnail"}),
    coordinates=Neo4jPoint(latitude=location.latitude, longitude=location.longitude)
)

property_on_sale_neo4j_db = PropertyOnSaleNeo4JDB(property_on_sale_neo4j)

# Create property on sale in Neo4j
neo4j_response = property_on_sale_neo4j_db.create_property_on_sale_neo4j(input_property_on_sale.neighbourhood)

# Update score
neo4j_response = property_on_sale_neo4j_db.update_livability_score()

return JSONResponse(
    status_code=201,
    content={
        "detail": "Property created successfully.",
        "property_on_sale_id": db_property_on_sale.property_on_sale.property_on_sale_id
    }
)
```

Figure 6.4: Create Property On Sale Controller (Code)

The sequence of operations is as follows:

- Authenticate the user via `verifyAccessToken`.
- Extract geographical coordinates from the property's address necessary for Neo4j through a geocoding-API.
- Insert the property into the MongoDB *PropertyOnSale* collection using `create_property_on_sale`.
- Duplicate the property information as an embedded document in the seller's collection via `insert_property_on_sale`.
- Insert the property into Neo4j using `create_property_on_sale_neo4j`.
- Calculate the livability score using `calculate_livability_score`.

In Neo4j, inserting a property is resource-intensive because it involves creating relationships between the new property and all nearby POIs and properties. The `create_property_on_sale_neo4j` function performs an optimized geospatial query using Neo4j's *POINT* object. This operation is executed only once per property (except in rare cases like an address change) to ensure that subsequent operations, like viewing nearby POIs or properties, are efficient.

```python
def create_property_on_sale_neo4j(self, neighbourhood_name: str):      VinStan1, 6 days ago • 🛠Logger, try catch + score
    neo4j_driver = get_neo4j_driver()
    if neo4j_driver is None:
        logger.error("Neo4j driver not initialized.")
        return 500
    try:
        with neo4j_driver.session() as session:
            def tx_func(tx):
                # Extract properties from the model dump and remove the 'coordinates' key from the properties dictionary.
                properties = self.property_on_sale_neo4j.model_dump()
                coordinates = properties.pop('coordinates', None)

                # Verify that the coordinates are provided.
                if coordinates is None:
                    raise ValueError("Coordinates were not provided")

                # Use MERGE to avoid duplicate nodes, and set the node properties.
                # The coordinates are set using the Neo4j built-in 'point' function.
                query1 = """
                MERGE (p:PropertyOnSale {property_on_sale_id: $property_on_sale_id})
                ON CREATE SET p += $properties_on_sale,
                              p.coordinates = point({latitude: $latitude, longitude: $longitude})
                ON MATCH SET p += $properties_on_sale,
                             p.coordinates = point({latitude: $latitude, longitude: $longitude})
                """
                tx.run(
                    query1,
                    property_on_sale_id=self.property_on_sale_neo4j.property_on_sale_id,
                    properties_on_sale=properties,
                    latitude=coordinates['latitude'],
                    longitude=coordinates['longitude']
                )

                # The neighbourhood is identified by its name.
                query2 = """
                MATCH (p:PropertyOnSale {property_on_sale_id: $property_on_sale_id})
                MERGE (n:Neighbourhood {name: $neighbourhood_name})
                MERGE (p)-[:LOCATED_IN_NEIGHBOURHOOD]->(n)
                """
```

```python
                tx.run(
                    query2,
                    property_on_sale_id=self.property_on_sale_neo4j.property_on_sale_id,
                    neighbourhood_name=neighbourhood_name
                )

                # Step 3: Create NEAR relationships between the property and all existing POI nodes
                # where the distance between their coordinates is less than or equal to 500 meters.
                query3 = """
                MATCH (p:PropertyOnSale {property_on_sale_id: $property_on_sale_id}), (poi:POI)
                WHERE point.distance(p.coordinates, poi.coordinates) <= 500
                MERGE (p)-[r:NEAR]->(poi)
                SET r.distance = point.distance(p.coordinates, poi.coordinates)
                """
                tx.run(
                    query3,
                    property_on_sale_id=self.property_on_sale_neo4j.property_on_sale_id
                )

                # Step 4: Create bidirectional NEAR_PROPERTY relationships between the new property and all existing properties
                # where the distance is less than or equal to 500 meters.
                query4 = """
                MATCH (p:PropertyOnSale {property_on_sale_id: $property_on_sale_id}), (other:PropertyOnSale)
                WHERE other.property_on_sale_id <> $property_on_sale_id
                AND point.distance(p.coordinates, other.coordinates) <= 500
                MERGE (p)-[r:NEAR_PROPERTY]->(other)
                SET r.distance = point.distance(p.coordinates, other.coordinates)
                MERGE (other)-[r2:NEAR_PROPERTY]->(p)
                SET r2.distance = point.distance(p.coordinates, other.coordinates)
                """
                tx.run(
                    query4,
                    property_on_sale_id=self.property_on_sale_neo4j.property_on_sale_id
                )
            # Execute all transactional steps
            session.write_transaction(tx_func)
        return 201
    except Exception as e:
        logger.error("Error while creating property on sale on Neo4j with id %s: %s",
                     self.property_on_sale_neo4j.property_on_sale_id, e)
        return 500
```

Data insertion in Neo4j is transaction-based, ensuring data consistency. The use of

transactions is justified by the infrequent nature of property insertions (each seller typically adds a new property about once a week) and also by the importance of relationships in a graph-based database. The liveability score is computed during property creation and stored as a permanent field, as it is based on static factors (e.g., nearby schools, hospitals, parks), avoiding repetitive queries to calculate it.

Sell a Property for Sale (also for Neo4j and Redis)

This operation marks a property as sold, removing it from the available listings. It is managed by the REST controller:

```
@seller_router.post("/sell_property_on_sale", ...)
```

which, after authenticating the user, calls the necessary methods to update all databases maintain data consistency.

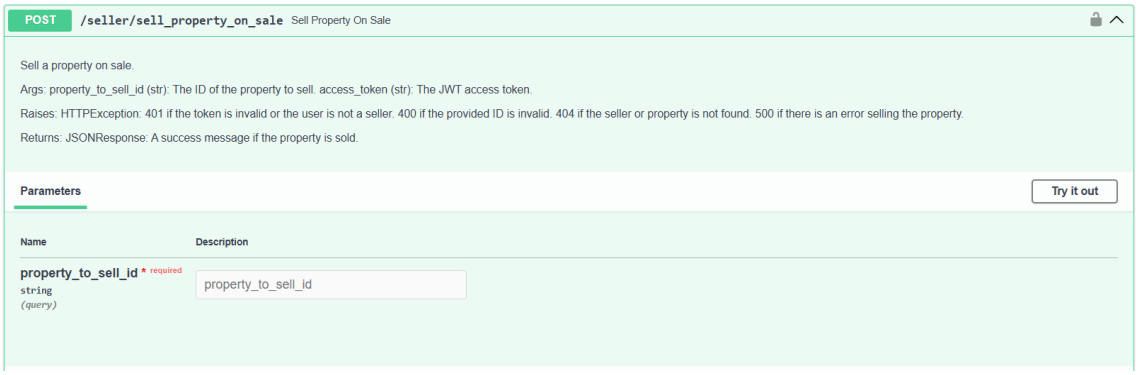

Figure 6.5: Sell Property On Sale Controller

The sequence is as follows:

- Authenticate the user via `verifyAccessToken`.
- Verify property ownership using `check_property_on_sale`.
- Retrieve property data to create an embedded document (representing the sold property) from the seller's collection while deleting the original document from the *PropertyOnSale* collection via `delete_and_return_property`.
- Update the seller's collection by calling `sell_property`, which removes the property from the on-sale list and adds it to the sold list.
- Ensure consistency in Redis by using `handleReservations`.
- Update Neo4j using `delete_property_on_sale_neo4j`.

The Redis reservation system relies on the function `handleReservation`. It is crucial that this function does not fail; otherwise, the *ReservationsBuyer* may become

inconsistent, leading to significant issues; for example, a user might attend an open house event for a property that has already been sold.

```python
def handleReservations(property_on_sale_id: str) -> int:
    reservation_seller = ReservationsSeller(property_on_sale_id=property_on_sale_id)
    reservation_seller_db = ReservationsSellerDB(reservation_seller)
    status = reservation_seller_db.get_reservation_seller()
    if status == 500:
        return 500
    buyer_ids = [reservation.buyer_id for reservation in reservation_seller_db.reservations_seller.reservations]

    not_deleted_ids = []
    for buyer_id in buyer_ids:
        reservation_buyer = ReservationsBuyer(buyer_id=buyer_id)
        reservation_buyer_db = ReservationsBuyerDB(reservation_buyer)
        status = reservation_buyer_db.delete_reservation_by_property_on_sale_id(property_on_sale_id)
        if status == 200:
            buyer_ids.remove(buyer_id)
        else:
            not_deleted_ids.append(buyer_id)

    if not_deleted_ids:
        run_date = datetime.now() + timedelta(minutes=5)
        scheduler.add_job(
            check_and_delete_when_low_load,
            'date',
            run_date=run_date,
            args=[not_deleted_ids, property_on_sale_id]
        )
    return 200
```

If an error occurs during the execution of `handleReservation`, the function is rescheduled to run again after 5 minutes, attempting to cancel all reservations for each buyer who has scheduled an open house event. If the function fails during execution, it is retried for the missing keys; however, to improve performance and avoid executing heavy operations when the servers are busy, this retry is performed only if the CPU load is low, as determined by the `check_and_delete_when_low_load` function.

```python
def check_and_delete_when_low_load(not_deleted_ids: list, property_on_sale_id: str):
    # psutil.cpu_percent() checks the current CPU load
    current_load = psutil.cpu_percent(interval=1)
    if current_load < 30:
        # If the load is low, update the reservations
        for buyer_id in not_deleted_ids:
            reservation_buyer = ReservationsBuyer(buyer_id=buyer_id)
            reservation_buyer_db = ReservationsBuyerDB(reservation_buyer)
            status = reservation_buyer_db.get_reservations_by_user()
            if status != 200:
                continue
            status = reservation_buyer_db.delete_reservation_by_property_on_sale_id(property_on_sale_id)
            # If the delete do not fail, remove the buyer_id from the list
            if status == 200:
                not_deleted_ids.remove(buyer_id)

        if not_deleted_ids:
            # If there are still reservations to delete, re-schedule the delete
            next_run = datetime.now() + timedelta(minutes=5)
            scheduler.add_job(
                check_and_delete_when_low_load,
                'date',
                run_date=next_run,
                args=[not_deleted_ids, property_on_sale_id]
            )
    else:
        # If the load is still high, re-schedule the delete
        next_run = datetime.now() + timedelta(minutes=5)
        scheduler.add_job(
            check_and_delete_when_low_load,
            'date',
            run_date=next_run,
            args=[not_deleted_ids, property_on_sale_id]
        )
```

Analytic 2

This analytic is part of the seller's activity monitoring service. It calculates the number of houses sold and the revenue generated in a given city, grouped by neighbourhood and within a specified time period. This information helps real estate agencies monitor productivity, identify trends, and develop marketing strategies.

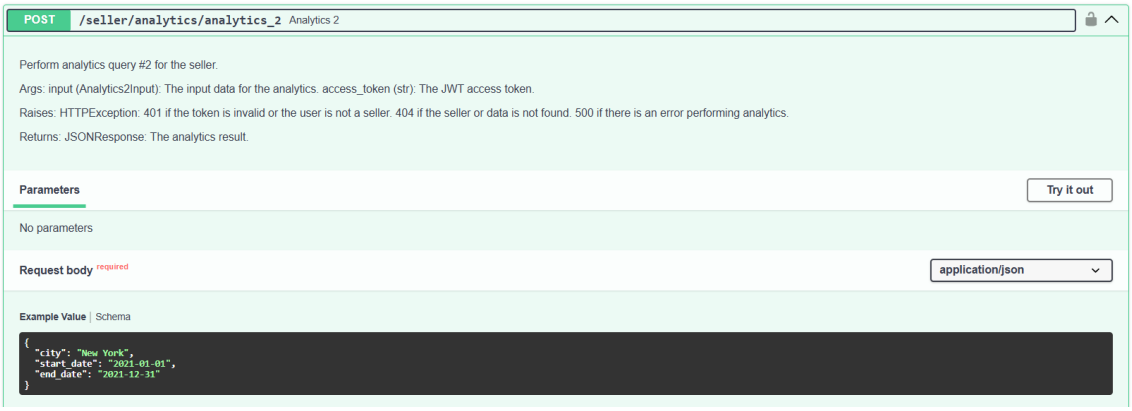

Figure 6.6: Analytic 2 Controller

The method validates the input, constructs an aggregation pipeline on the *seller* collection that:

- Identifies the seller requesting the analytic.
- Collects houses sold by that seller.
- Filters houses based on user-defined parameters.
- Groups the data by neighbourhood and computes the count and total revenue.
- Renames the attributes accordingly.

Analytic 3

This analytic provides, for a given city and time period, the average selling time (in days) and the total number of houses sold, broken down by neighbourhood. A house is included only if it was both registered and sold within the specified interval.

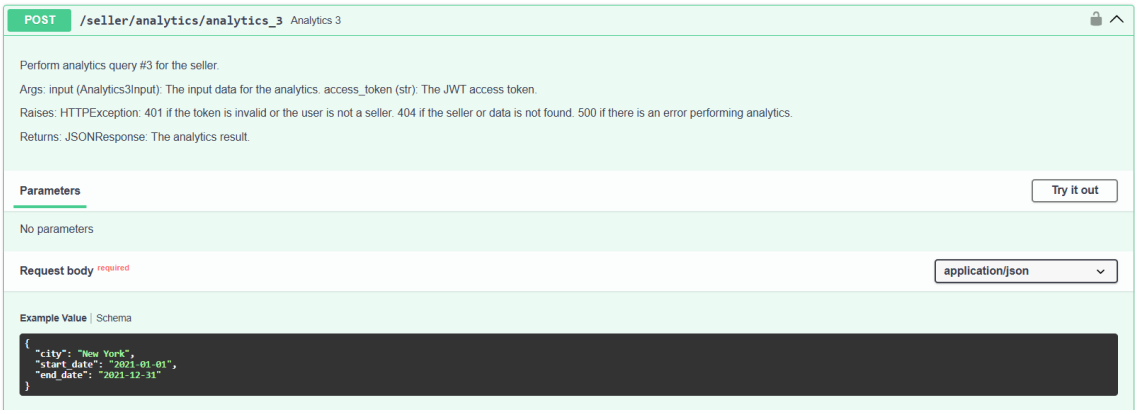

Figure 6.7: Analytic 3 Controller

After input validation, the method constructs an aggregation pipeline on the *seller* collection:

- Identify the seller.
- Create a collection of houses sold by the seller.
- Filter houses based on user-specified criteria.
- Compute a new attribute for each house calculating the selling time in days:

$$\frac{\text{sale\_date} - \text{registration\_date}}{\text{milliseconds\_per\_day}}$$

- Group by neighbourhood to compute the average selling time and the number of houses sold.
- Rename attributes accordingly.

6.1.3 All Users

Filtered Search

This operation allows any user to retrieve properties for sale that match selected criteria. The available filters include:

- City
- Address (checked with `RegEx`)
- Neighbourhood
- Maximum price
- Property type
- Minimum area
- Minimum number of bedrooms
- Minimum number of bathrooms

It is handled by the REST controller:

```
@guest_router.post("/properties_on_sale/search", ...)
```

which calls the `filtered_search` method of the *PropertyOnSaleDB* class. This method dynamically constructs a query for MongoDB's `find` operation, based on the selected parameters and returns a list of matching properties via the attribute `property_on_sale_list`. In order to make the operation fast, only summary attributes are transmitted for each property. Then, if a user wants to see all information about a property, can request the operation *See property page*, that executes a MongoDB `find` with the `property_on_sale_id` and retrieves all the missing information.

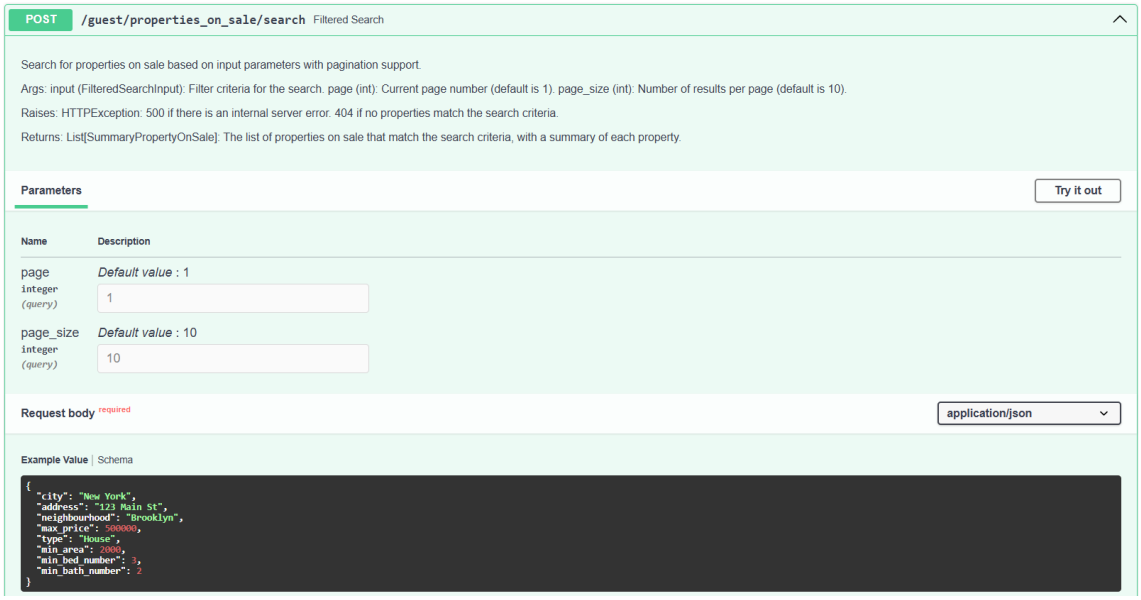

Figure 6.8: Filtered Search Controller

```python
def filtered_search(self, input: FilteredSearchInput, page: int = 1, page_size: int = 10) -> int:
    mongo_client = get_default_mongo_db()
    if mongo_client is None:
        logger.error("Mongo client not initialized.")
        return 500

    query = {}
    if input.city:
        query["city"] = input.city
    if input.address:
        query["address"] = {"$regex": input.address, "$options": "i"}
    if input.max_price:
        query["price"] = {"$lte": input.max_price}
    if input.neighbourhood:
        query["neighbourhood"] = input.neighbourhood
    if input.type:
        query["type"] = input.type
    if input.min_area:
        query["area"] = {"$gte": input.min_area}
    if input.min_bed_number:
        query["bed_number"] = {"$gte": input.min_bed_number}
    if input.min_bath_number:
        query["bath_number"] = {"$gte": input.min_bath_number}
```

```
try:
    # Initialize the cursor with the query and projection
    results_cursor = mongo_client.PropertyOnSale.find(
    query,
    {
        "_id": 1,
        "type": 1,
        "address": 1,
        "thumbnail": 1,
        "price": 1,
        "registration_date": 1,
        "city": 1,
        "neighbourhood": 1,
        "area" :1
    }
    )
    # Apply pagination using skip and limit
    skip = (page - 1) * page_size
    results_cursor = results_cursor.skip(skip).limit(page_size)
    results_list = list(results_cursor)
except Exception as e:
    logger.error("Error during search: %s", e)
    return 500

if not results_list:
    return 404

self.property_on_sale_list = []
for result in results_list:
    result["property_on_sale_id"] = str(result["_id"])
    self.property_on_sale_list.append(PropertyOnSale(**result))
return 200
```

6.1.4 Registered Users

Analytics 1

This analytic is part of the market monitoring functionality. For a given city, it calculates the average selling price per square meter (optionally filtered by property type) grouped by neighbourhood. This provides buyers insight into premium areas and helps sellers set competitive prices.

It is managed by the REST controller:

```
@registered_user_router.post("/analytics_1", ...)
```

along with `verifyAccessToken` and `get_avg_price_per_square_meter` from the *PropertyOnSale* class.

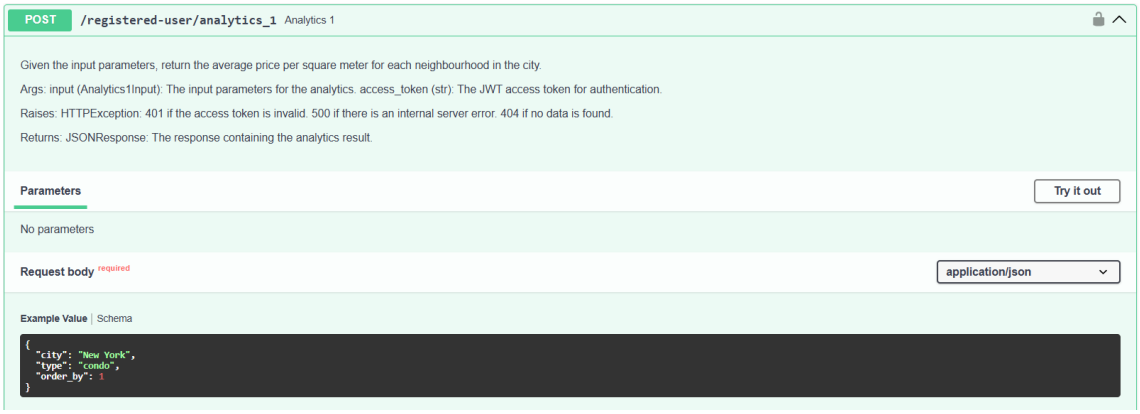

Figure 6.9: Analytic 1 Controller

The method constructs and executes an aggregation pipeline on the *PropertyOnSale* collection:

- Filter properties based on user input (with an optional property type filter).
- Calculate the price per square meter for each property.
- Group by neighbourhood to compute the average price per square meter.
- Rename the attributes accordingly.
- Sort the results based on the user's request.

Analytics 5

This analytic provides, for a given city and neighbourhood, the average number of bedrooms, bathrooms, and the average area per property type. It is designed primarily for buyers to understand typical property sizes across different property types. (Note: Analytics 4, which calculates the average price per property type, follows similar logic and is not detailed here.)

The operation is managed by the REST controller:

```
@registered_user_router.post("/analytics_5", ...)
```

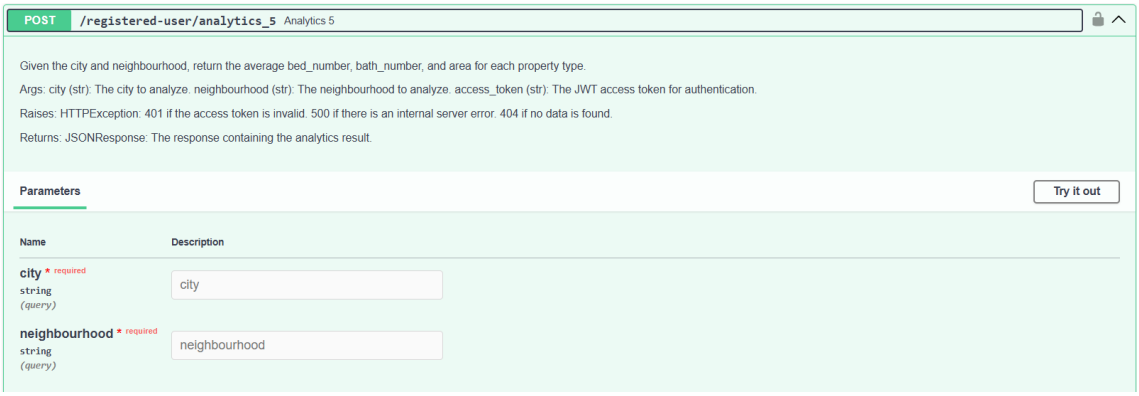

Figure 6.10: Analytic 5 Controller

After authentication, the controller calls `get_statistics_by_city_and_neighbourhood` of the *PropertyOnSale* class. This method constructs a pipeline on the *PropertyOnSale* collection:

- Filter properties by city and neighbourhood.
- Group by property type and compute the required aggregations.
- Rename attributes accordingly.

6.2 Redis

6.2.1 Buyer

View Next Appointments

This operation allows a buyer to view all upcoming open house events for which they are registered. Each event displays:

- Date
- Time slot
- Address
- Property thumbnail

If the buyer wishes to view more details about a property, clicking the reservation redirects them to the property page.

It is managed by the REST controller:

```
@buyer_router.get("/reservations", ...)
```

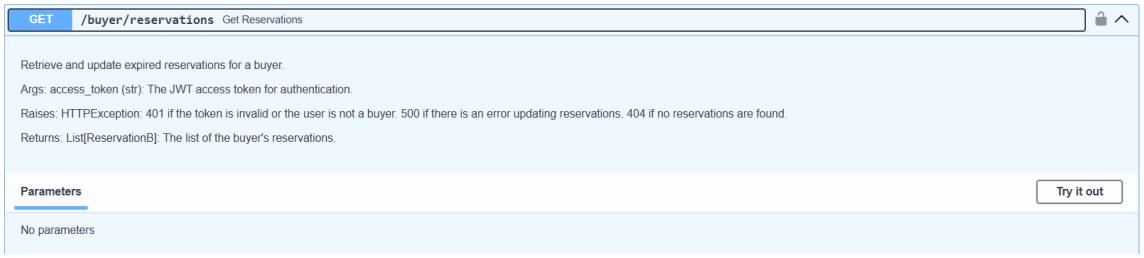

Figure 6.11: Get Reservations Controller

After authentication, the controller calls `update_expired_reservations` from the *ReservationsBuyerDB* class, which interacts with the Redis store to retrieve and update the buyer's reservations.

```python
def update_reservation_seller(self, buyer_id: str, updated_data: dict) -> int:
    redis_client = get_redis_client()
    if redis_client is None:
        logger.error("Failed to connect to Redis.")
        return 500
    key = f"property_on_sale_id:{self.reservations_seller.property_on_sale_id}:reservations_seller"
    raw_data = redis_client.get(key)
    if not raw_data:
        return 404
    try:
        data = json.loads(raw_data)
        updated = False
        for i, item in enumerate(data):
            if item.get("buyer_id") == buyer_id:
                data[i].update(updated_data)
                updated = True
                break
        if not updated:
            return 404
        return 200
    except (json.JSONDecodeError, TypeError, redis.exceptions.RedisError) as e:
        logger.error(f"Error updating seller reservation with buyer_id={buyer_id}: {e}")
        return 500
```

The method constructs a search key using the `buyer_id` and, with a single `GET` operation, retrieves the entire list of reservations for that buyer. It then iterates through the list to check for expired reservations using the `check_reservation_expired` method. If any expired reservations are found, the method updates the key-value database with the revised list. This approach eliminates the need for periodic scans of all key-value database entries to check for expired reservations. Consequently, the cost of viewing a buyer's reservations remains constant even when the operation is requested multiple times in a day. The design, including the slight overhead due to data replication in Redis, is justified by the optimization it provides for this functionality.

Make a Reservation

This critical operation ensures efficient reservation management on both the buyer and seller sides. It is handled by the REST controller:

```
@buyer_router.post("/reservation", ...)
```

Figure 6.12: Create Reservation Controller

```python
@buyer_router.post(
    "/reservation",
    response_model=ResponseModels.SuccessModel,
    responses=ResponseModels.CreateReservationBuyerResponses
)
def create_reservation(book_now_info: CreateReservationBuyer, access_token: str = Depends(JWTHandler())):
    buyer_id, user_type = JWTHandler.verifyAccessToken(access_token)
    if buyer_id is None or user_type != "buyer":
        raise HTTPException(status_code=401, detail="Invalid access token")

    buyer = Buyer(buyer_id=buyer_id)
    buyer_db = BuyerDB(buyer)

    status = buyer_db.get_profile_info()
    if status == 404:
        raise HTTPException(status_code=404, detail="Buyer not found")
    if status == 500:
        raise HTTPException(status_code=500, detail="Error decoding buyer data")

    buyer = buyer_db.buyer
    if not all([buyer.name, buyer.surname, buyer.email, buyer.phone_number]):
        raise HTTPException(status_code=500, detail="Incomplete buyer data")

    reservations_buyer = ReservationsBuyer(buyer_id=buyer_id)
    reservations_buyer_db = ReservationsBuyerDB(reservations_buyer)

    status = reservations_buyer_db.get_reservations_by_user()
    if status == 500:
        raise HTTPException(status_code=500, detail="Error decoding reservations data")
    if status == 400:
        raise HTTPException(status_code=400, detail="Invalid input data")
    #check if the buyer already has a reservation for the same property
    if reservations_buyer_db.reservations_buyer.reservations:
        for res in reservations_buyer_db.reservations_buyer.reservations:
            if res.property_on_sale_id == book_now_info.property_on_sale_id:
                raise HTTPException(status_code=409, detail="Reservation already exists")

    reservations_seller = ReservationsSeller(property_on_sale_id=book_now_info.property_on_sale_id)
    reservations_seller_db = ReservationsSellerDB(reservations_seller)
```

```python
    new_reservation = ReservationS(
        buyer_id=buyer_id,
        full_name=f"{buyer.name} {buyer.surname}",
        email=buyer.email,
        phone=buyer.phone_number
    )

    status = reservations_seller_db.handle_book_now_transaction(new_reservation, book_now_info.day, book_now_info.time, buyer_id, book_now_info.max_attendees)
    if status == 400:
        raise HTTPException(status_code=400, detail="Invalid input data")
    if status == 500:
        raise HTTPException(status_code=500, detail="Error creating reservation")

    date = next_weekday(book_now_info.day)

    reservations_buyer_db.reservations_buyer.reservations = [
        ReservationB(
            property_on_sale_id = book_now_info.property_on_sale_id,
            date = date,
            time = book_now_info.time,
            thumbnail = book_now_info.thumbnail,
            address = book_now_info.address
        )
    ]
    status = reservations_buyer_db.create_reservation_buyer()

    # If error occurs, rollback the reservation
    if status != 201:
        reservations_seller_db.delete_reservation_seller_by_buyer_id(buyer_id)
        return JSONResponse(status_code=500, content={"detail": "Error creating reservation"})

    return JSONResponse(status_code=201, content={"detail": "Reservation created successfully"})
```

Figure 6.13: Create Reservation Controller (Code)

It takes as input the data necessary to create reservations on both sides: property_on_sale_id, day, time, max_reservation, thumbnail, address. These data are already available on the client because, before performing this operation, the user visited the page of a property (through *See property page*), receiving all its information in response to the `GET` request.

which performs the following steps:

- Use `get_profile_info` to fetch buyer details.
- Call `get_reservations_by_user` to check for an existing reservation for the same property in the buyer reservations list.
- If no reservation exists, call `handle_book_now_transaction` to insert the reservation on the seller side, this function ensures also that the maximum number of reservations is not exceeded.
- Finally, call `create_reservation_buyer` to record the reservation on the buyer side with the information needed.

```python
def handle_book_now_transaction(self, reservation: ReservationS, day: str, time: str, buyer_id: str, max_attendees: int) -> int:
    redis_client = get_redis_client()
    if redis_client is None:
        logger.error("Failed to connect to Redis.")
        return 500
    key = f"property_on_sale_id:{self.reservations_seller.property_on_sale_id}:reservations_seller"
    ttl = convert_to_seconds(day, time)
    if ttl is None:
        return 400

    try:
        with redis_client.pipeline() as pipe:
            # Watch the key to detect concurrent modifications
            pipe.watch(key)

            # Retrieve the data BEFORE starting the transaction
            raw_data = pipe.get(key)
            reservations_list = json.loads(raw_data) if raw_data else []

            # Check max attendees
            if len(reservations_list) >= max_attendees:
                pipe.unwatch()
                return 400

            # Append new reservation
            reservations_list.append(reservation.model_dump())

            # Start transaction
            pipe.multi()
            pipe.setex(key, ttl, json.dumps(reservations_list))
            pipe.execute()
            return 200
    except redis.exceptions.WatchError:
        logger.error(f"Transaction conflict on key={key}.")
        return 500
    except (redis.exceptions.RedisError, json.JSONDecodeError) as e:
        logger.error(f"Error in handle_book_now_transaction: {e}")
        return 500
```

The `handle_book_now_transaction` method creates a JSON document containing the buyer's contact information and uses Redis's `WATCH` operation to ensure an atomic transaction. If the reservation list doesn't exist, it is created with a TTL based on the open house date.

6.2.2 Seller

View Reservations for an Open House Event

This operation allows a seller to view the contact details of all buyers registered for an open house event. The computational cost of this operation is $\mathcal{O}(1)$; the key used for the `GET` operation is the property *_id*. This query structure is essential because the seller's reservation data is always accessed contextually. Sellers are expected to perform this operation multiple times during a session, particularly for properties with an upcoming open house, so it must be optimized.

It is managed by the REST controller:

```
@seller_router.get("/reservations", ...)
```

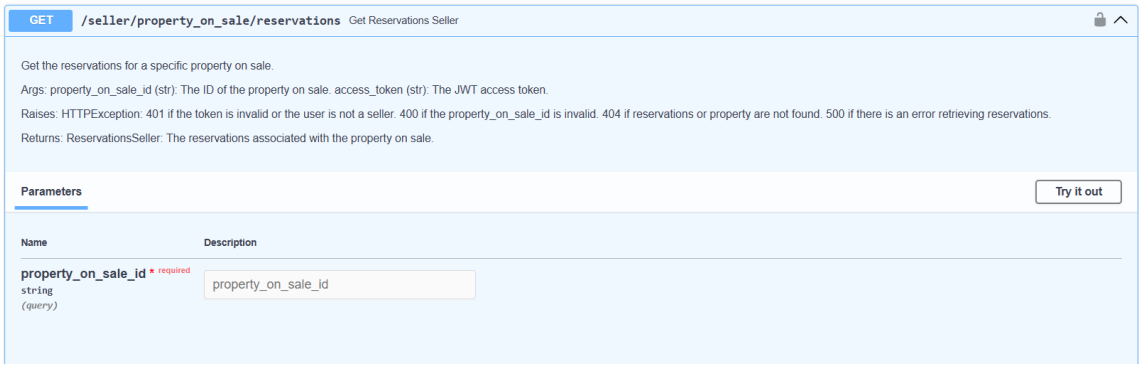

Figure 6.14: Get Reservations Seller Controller

which calls `get_reservation_seller` from the *ReservationsSellerDB* class. This method retrieves buyer contact details using the property *_id*.

6.3 Neo4j

Suggests Houses

This operation provides house suggestions by traversing the graph. It is managed by the REST controller:

```
@guest_router.get("/map/properties_near_property", ...)
```

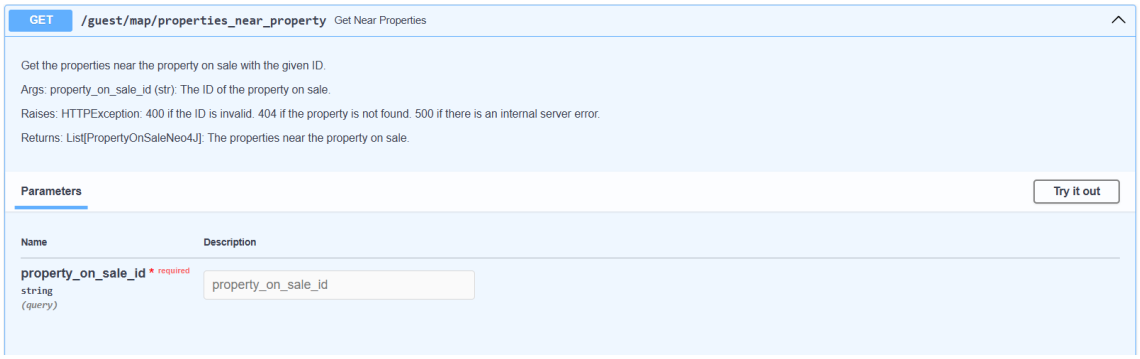

Figure 6.15: Get Near Properties Controller

which calls `get_near_properties` from the *PropertyOnSaleNeo4JDB* class. The method traverses nodes of type `PropertyOnSale` connected via the `PROPERTY_NEAR_PROPERTY` relationship up to 2 hops from the target property.

```python
def get_near_properties(self):
    neo4j_driver = get_neo4j_driver()
    if neo4j_driver is None:
        logger.error("Neo4j driver not initialized.")
        return 500
    with neo4j_driver.session() as session:
        try:
            query = """
            MATCH (p:PropertyOnSale {property_on_sale_id: $property_on_sale_id})
            OPTIONAL MATCH (p)-[:NEAR_PROPERTY]->(p2:PropertyOnSale)
            OPTIONAL MATCH (p2)-[:NEAR_PROPERTY]->(p3:PropertyOnSale)
            WITH collect(p) AS pList, collect(DISTINCT p2) AS level1, collect(DISTINCT p3) AS level2
            WITH pList + level1 + level2 AS allNodes
            UNWIND allNodes AS n
            RETURN collect(DISTINCT n) AS uniqueNodes
            """
            result = session.run(query, property_on_sale_id=self.property_on_sale_neo4j.property_on_sale_id)
        except Exception as e:
            logger.error("Error while retrieving near properties to property %s: %s",
                         self.property_on_sale_neo4j.property_on_sale_id, e)
            return 500

        record = result.single()
        if record is None:
            return 404

        # Retrieve the unique nodes from the query result (returned as "uniqueNodes")
        unique_nodes = record.get("uniqueNodes", [])

        self.near_properties = []
        for node in unique_nodes:
            # Create a Neo4jPoint object from the node's coordinates
            coordinates = Neo4jPoint(latitude=node["coordinates"].latitude, longitude=node["coordinates"].longitude)
            # Build the PropertyOnSaleNeo4J object, excluding the 'coordinates' property from the node dictionary,
            # and adding the serialized coordinates.
            self.near_properties.append(
                PropertyOnSaleNeo4J(
                    **{k: v for k, v in dict(node).items() if k != "coordinates"},
                    coordinates=coordinates.model_dump()
                )
            )

        return 200
```

View Points of Interest

This operation retrieves points of interest near a property. Managed by the REST controller:

```
@guest_router.get("/map/pois_near_property", ...)
```

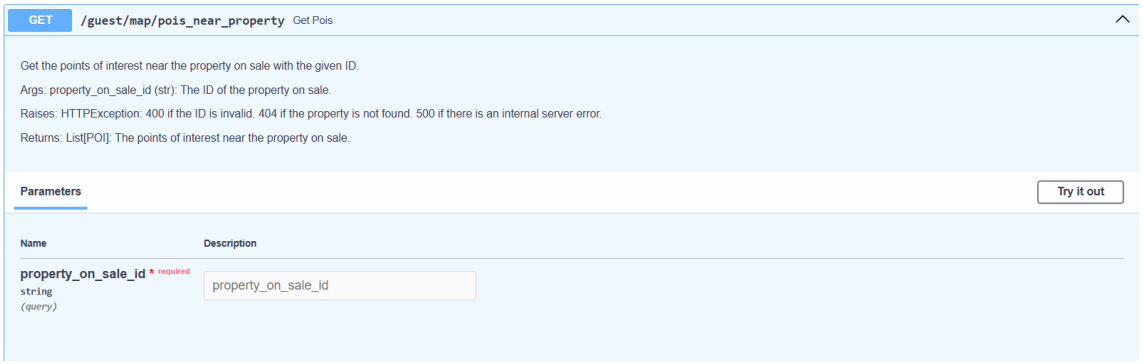

Figure 6.16: Get POIs Controller

it calls `get_near_POIs` from the *PropertyOnSaleNeo4JDB* class. The method traverses nodes of type `POI` linked by the `PROPERTY_NEAR_POI` relationship.

Learn about the Neighbourhood and City Where a House is Located

This operation retrieves information about the neighbourhood and city of a property. It is managed by the REST controller:

```
@guest_router.get("/map/city_and_neighborhood", ...)
```

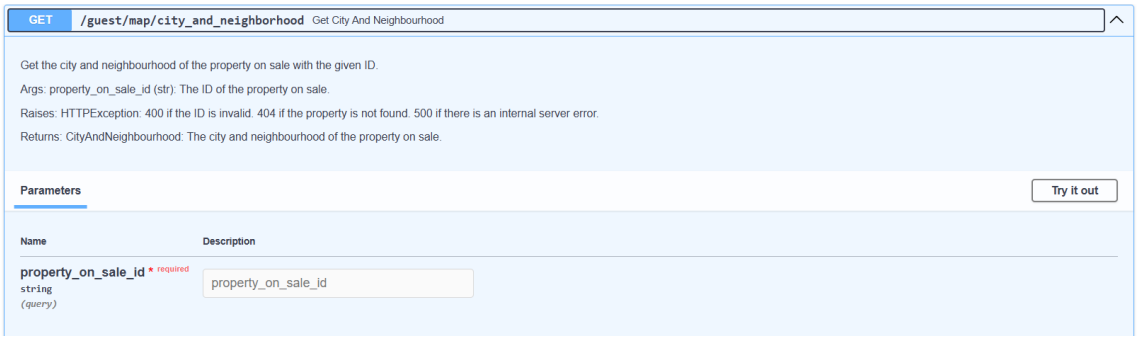

Figure 6.17: Get City And Neighbourhood Controller

which calls `get_city_and_neighbourhood` from the *PropertyOnSaleNeo4JDB* class. The method traverses the graph from the property node, making up to 2 hops using the relationships `NEIGHBOURHOOD_BELONGS_TO_CITY` and `PROPERTY_LOCATED_IN_NEIGHBOURHOOD`.

```python
def get_city_and_neighbourhood(self):
    neo4j_driver = get_neo4j_driver()
    if neo4j_driver is None:
        logger.error("Neo4j driver not initialized.")
        return 500
    print(self.property_on_sale_neo4j.property_on_sale_id)
    with neo4j_driver.session() as session:
        try:
            query = """
            MATCH (p:PropertyOnSale {property_on_sale_id: $property_on_sale_id})
            MATCH (p)-[:LOCATED_IN_NEIGHBOURHOOD]->(n:Neighbourhood)-[:BELONGS_TO_CITY]-(c:City)
            RETURN n AS neighbourhood, c AS city
            """
            result = session.run(query, property_on_sale_id=self.property_on_sale_neo4j.property_on_sale_id)
        except Exception as e:
            logger.error("Error while retrieving city and neighbourhood for property %s: %s",
                         self.property_on_sale_neo4j.property_on_sale_id, e)
            return 500
        result_list = list(result)
        if not result_list:
            return 404
        row = result_list[0]
        city_coordinates = Neo4jPoint(latitude=dict(row["city"])["coordinates"].latitude, longitude=dict(row["city"])["coordinates"].longitude)
        neighbourhood_coordinates = Neo4jPoint(latitude=dict(row["neighbourhood"])["coordinates"].latitude, longitude=dict(row["neighbourhood"])["coordinates"].longitude)
        self.city = City(**{k: v for k, v in dict(row["city"]).items() if k != 'coordinates'}, coordinates=city_coordinates.model_dump())
        self.neighbourhood = Neighbourhood(**{k: v for k, v in dict(row["neighbourhood"]).items() if k != 'coordinates'}, coordinates=neighbourhood_coordinates.model_dump())
        return 200
```

6.3.1 Translation from Domain-Centric Queries to Graph-Centric Queries

Domain-Centric	Graph-Centric
Get properties near a property with the given ID	Get `PropertyOnSale` nodes which are one or two hops away from a certain `PropertyOnSale` node, considering the relationship `NEAR_PROPERTY`
Get POIs near a property with the given ID	Get `POIs` nodes which are one hop away from a certain `PropertyOnSale` node, considering the relationship `NEAR`
Get the City and Neighbourhood of a property with the given ID	Get `City` and `Neighbourhood` nodes of a certain `PropertyOnSale` node, through the relationships `LOCATED_IN_NEIGHBOURHOOD` and `BELONGS_TO_CITY`
Calculate the livability index	Get `POIs` nodes which are one hop away from a certain `PropertyOnSale` node, considering the relationship `NEAR` and counting how many for each `type` are present

Table 6.1: Translation from Domain-Centric to Graph-Centric

6.4 Queries in Native Language

6.4.1 MongoDB

Analytics 1

```
db.PropertyOnSale.aggregate ([
  {
    "$match": {
      "model_dump": { "city" : "input_city", "type": "
   optional_input_type" }
    }
  },
  {
    "$project": {
      "neighbourhood": 1,
      "price_per_square_meter": { "$divide": ["$price", "
   $area"] }
    }
  },
  {
    "$group": {
      "_id": "$neighbourhood",
      "avg_price": { "$avg": "$price_per_square_meter" }
    }
  },
  {
    "$project": {
      "neighbourhood": "$_id",
      "avg_price": 1,
      "_id": 0
    }
  },
  {
    "$sort": {
      "avg_price": input_order
    }
  }
])
```

Analytics 4

```
db.PropertyOnSale.aggregate ([
  {
```

```
    "$match": {
      "city": "input_city"
    }
  },
  {
    "$project": {
      "type": 1,
      "price": 1
    }
  },
  {
    "$group": {
      "_id": "$type",
      "avg_price": { "$avg": "$price" },
      "count": { "$sum": 1 }
    }
  },
  {
    "$project": {
      "type": "$_id",
      "avg_price": 1,
      "_id": 0,
      "count": 1
    }
  }
])
```

Analytics 5

```
db.PropertyOnSale.aggregate ([
  {
    "$match": {
      "city": "Input_city",
      "neighbourhood": "Input_neighbourhood"
    }
  },
  {
    "$group": {
      "_id": "$type",
      "avg_bed_number": { "$avg": "$bed_number" },
      "avg_bath_number": { "$avg": "$bath_number" },
      "avg_area": { "$avg": "$area" }
    }
  },
  {
```

```
    "$project": {
      "type": "$_id",
      "avg_bed_number": 1,
      "avg_bath_number": 1,
      "avg_area": 1,
      "_id": 0
    }
  }
])
```

Analytics 2

```
db.Seller.aggregate([
  {
    "$match": {
      "_id": "Input_id"
    }
  },
  {
    "$unwind": "$sold_properties"
  },
  {
    "$match": {
      "sold_properties.city": "Input_city",
      "sold_properties.sell_date": { "$gte": "
   Input_start_date", "$lte": "Input_end_date" }
    }
  },
  {
    "$group": {
      "_id": "$sold_properties.neighbourhood",
      "houses_sold": { "$sum": 1 },
      "revenue": { "$sum": "$sold_properties.price" }
    }
  },
  {
    "$project": {
      "neighbourhood": "$_id",
      "houses_sold": 1,
      "revenue": 1,
      "_id": 0
    }
  }
])
```

Analytics 3

```
db.Seller.aggregate([
  {
    "$match": {
      "_id": "Input_id"
    }
  },
  {
    "$unwind": "$sold_properties"
  },
  {
    "$match": {
      "sold_properties.city": "input.city",
      "sold_properties.sell_date": { "$gte": "
   Input_start_date", "$lte": "Input_end_date" },
      "sold_properties.registration_date": { "$gte": "
   Input_start_date", "$lte": "Input_end_date" }
    }
  },
  {
    "$project": {
      "time_to_sell": {
        "$divide": [
          { "$subtract": ["$sold_properties.sell_date", "
   $sold_properties.registration_date"] },
          86400000
        ]
      },
      "sold_properties.neighbourhood": 1
    }
  },
  {
    "$group": {
      "_id": "$sold_properties.neighbourhood",
      "avg_time_to_sell": { "$avg": "$time_to_sell" },
      "num_house": { "$sum": 1 }
    }
  },
  {
    "$project": {
      "neighbourhood": "$_id",
      "avg_time_to_sell": 1,
      "num_house": 1,
      "_id": 0
    }
  }
```

```
])
```

6.4.2 Redis

Create Reservation (Buyer)

```
    GET buyer_id:<buyer_id>:reservations_buyer
    WATCH property_on_sale_id:<property_on_sale_id>:
  reservations_seller
    GET property_on_sale_id:<property_on_sale_id>:
  reservations_seller
    MULTI
    SETEX property_on_sale_id:<property_on_sale_id>:
  reservations_seller <ttl_in_seconds> <new_data>
    EXEC
    SET buyer_id:<buyer_id>:reservations_buyer <new_data>
```

Get Reservation (Buyer)

```
    GET buyer_id:{buyer_id}:reservations_buyer
    # Performed only if some reservations have expired
    SET buyer_id:{buyer_id}:reservations_buyer <
  old_key_updated>
```

Delete Reservation (Buyer)

```
    GET buyer_id:{buyer_id}:reservations_buyer
    GET property_on_sale_id:{property_on_sale_id}:
  reservations_seller
    GET buyer_id:{buyer_id}:reservations_buyer
    SET buyer_id:{buyer_id}:reservations_buyer <new_data>
    GET property_on_sale_id:{property_on_sale_id}:
  reservations_seller
    SET property_on_sale_id:{property_on_sale_id}:
  reservations_seller <new_data>
```

6.4.3 Neo4j

Get City and Neighbourhood

```
 MATCH (p:PropertyOnSale {property_on_sale_id:
$property_on_sale_id})
 MATCH (p)-[:LOCATED_IN_NEIGHBOURHOOD]->(n:
Neighbourhood)-[:BELONGS_TO_CITY]-(c:City)
 RETURN n AS neighbourhood, c AS city
```

Get Near POIs

```
 MATCH (p:PropertyOnSale)-[:NEAR]->(poi:POI)
 WHERE p.property_on_sale_id = $property_on_sale_id
 RETURN poi
```

Get Near Properties

```
 MATCH (p:PropertyOnSale {property_on_sale_id:
$property_on_sale_id})
 MATCH (p)-[:LOCATED_IN_NEIGHBOURHOOD]->(n:
Neighbourhood)-[:BELONGS_TO_CITY]-(c:City)
 RETURN n AS neighbourhood, c AS city

 MATCH (p:PropertyOnSale)-[:NEAR]->(poi:POI)
 WHERE p.property_on_sale_id = $property_on_sale_id
 RETURN poi

 MATCH (p:PropertyOnSale {property_on_sale_id:
$property_on_sale_id})
 OPTIONAL MATCH (p)-[:NEAR_PROPERTY]->(p2:
PropertyOnSale)
 OPTIONAL MATCH (p2)-[:NEAR_PROPERTY]->(p3:
PropertyOnSale)
 WITH collect(p) AS pList, collect(DISTINCT p2) AS
level1, collect(DISTINCT p3) AS level2
 WITH pList + level1 + level2 AS allNodes
 UNWIND allNodes AS n
 RETURN collect(DISTINCT n) AS uniqueNodes
```

Get info to compute the Livability Score

```
    MATCH (p:PropertyOnSale {property_on_sale_id:
$property_id})- [r:NEAR] -> (poi:POI)
    RETURN toLower(trim(poi.type)) AS poi_type, count(poi
) AS cnt, min(r.distance) AS min_distance
```

Chapter 7

Design Choices

7.1 Indexes

7.1.1 **MongoDB - PropertyOnSale:** Index(city, neighbourhood) **and** Index(city, price)

As highlighted in the volume table, one of the main operations performed on the document database is the filtered search among all available properties. It is realistic to assume that a user is primarily interested in properties located in a specific city, making the city filter a common selection in most searches. Additionally, it is highly probable that users are interested not in all houses within a city, but only in those located in a specific neighbourhood or within a certain price range. However, we cannot assume that all three filters will always be used simultaneously.

The introduction of composite indexes on `(city, neighbourhood)` and `(city, price)`, as illustrated in the figures below, significantly reduces query execution time and the number of documents scanned. This reduction in operational complexity, achieved through indexing, is particularly evident when comparing performance before and after applying these indexes.

Figure 7.1: Filtered Search (before)

Figure 7.2: Filtered Search (after)

We have decided not to introduce an additional three-field index `(city, neighbourhood, price)` because it does not significantly reduce the number of documents scanned or the query execution time compared to the two-field indexes. Even as the application grows, the city filter remains the primary factor in reducing the number of documents processed by the search query.

Figure 7.3: Filtered Search (before)

Figure 7.4: Filtered Search (after)

7.1.2 MongoDB - Buyer: Index(email) and MongoDB - Seller: Index(email)

User authentication is performed once per session for both buyers and sellers. The authentication function must search for a specific email within either the buyers' or sellers' collection, depending on the type of user logging in.

Considering the volume table for this operation and anticipating future growth, an index on the `email` field significantly reduces database load. This improvement is demonstrated by a decrease in the number of documents scanned and a reduction in query execution time, as shown in the figures below.

Figure 7.5: Login (before)

Figure 7.6: Login (after)

Only the test on the buyers' collection is shown here as the test on sellers' collection provides similar results.

7.1.3 Neo4j - PropertyOnSaleNeo4J: property_on_sale_id

Given the importance of the "See property on the map" operation in terms of volume, it is essential to define an index on the `_id` attribute of the **Property** node in Neo4j. This is necessary because all queries and functionalities in Neo4j originate from a specific property whose `_id` is already known to the client.

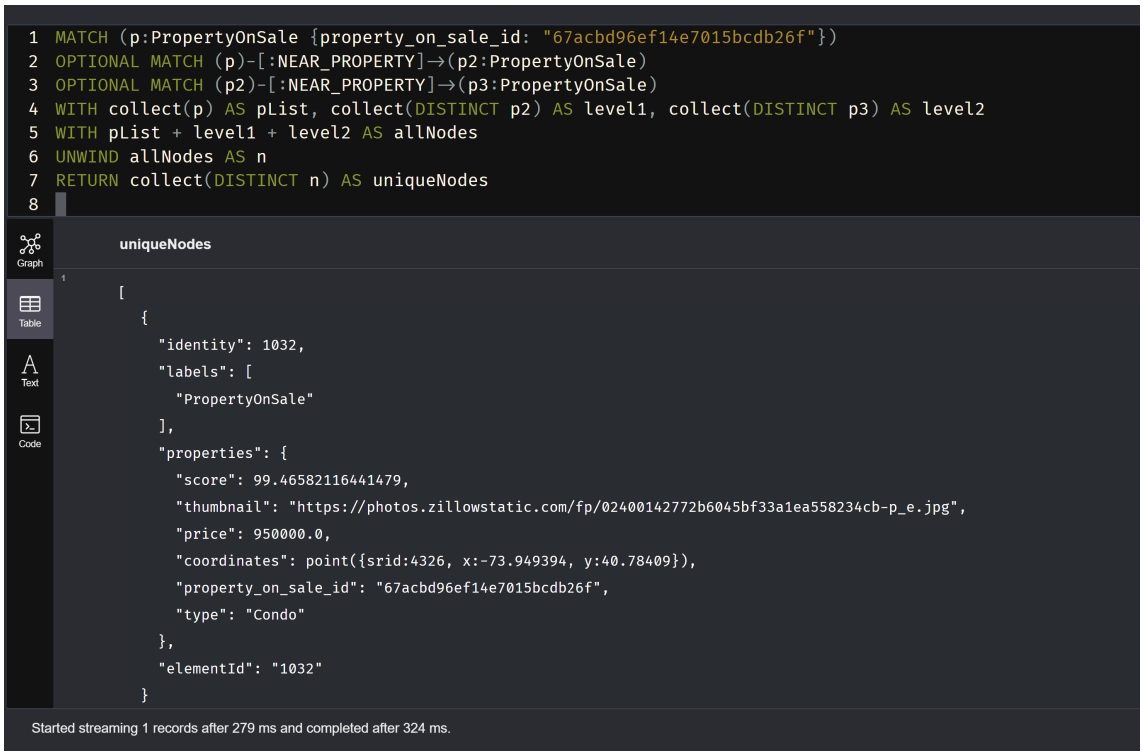

Figure 7.7: Get Near Properties (before)

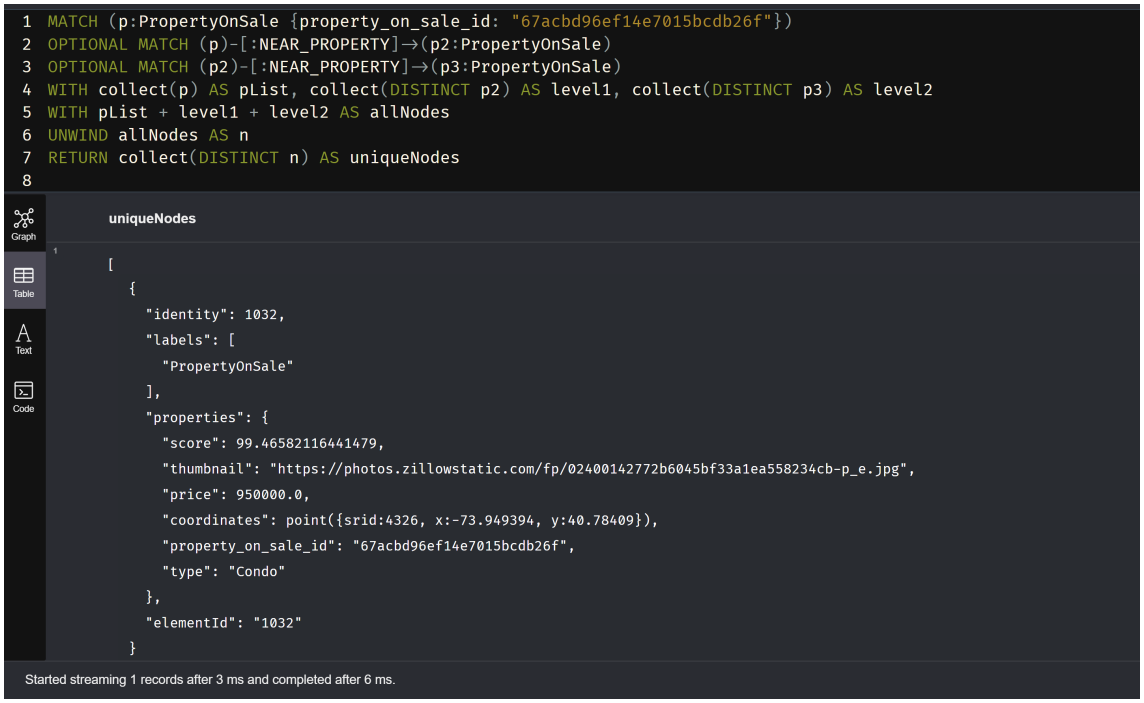

Figure 7.8: Get Near Properties (after)

7.1.4 Neo4j - Neighbourhood: name

An additional index on the **Neighbourhood** node was introduced in Neo4j to speed up the creation of `NEAR` relationships when inserting a new property. This index allows for faster retrieval of the neighbourhood node during the property insertion process, which is crucial for establishing relationships between the new property, its corresponding neighbourhood, and the city.

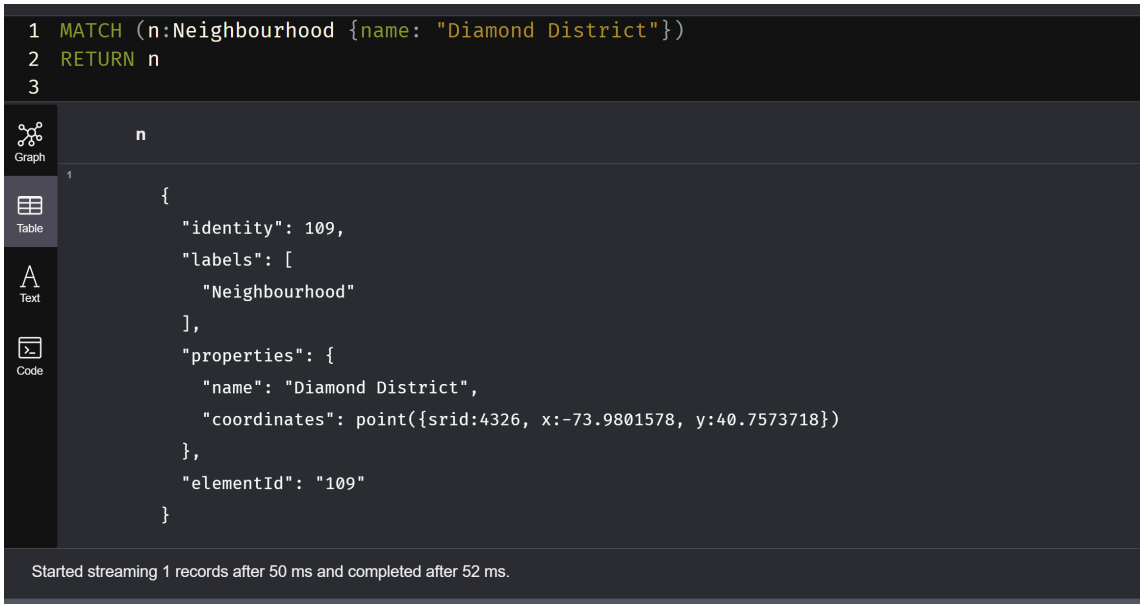

Figure 7.9: Get Neighbourhood (before)

Figure 7.10: Get Neighbourhood(after)

7.1.5 Consideration on Neo4j's Operations

During the creation of `NEAR` relationships between a **Property** node and another **Property** node, or between a **Property** node and a **POI** node, Neo4j leverages built-in indexes for the `POINT` data type. In particular, these indexes optimize the `point.distance` function. As a result, there is no need for additional optimization for these operations, as they are managed internally by Neo4j. Since these operations are not performed frequently, their computational complexity remains acceptable.

7.2 Sharding

7.2.1 Sharding on MongoDB

Sharding has been designed by selecting appropriate sharding keys and strategies while considering how queries would be affected.

PropertyOnSale

This collection is used for the following queries:

- Find a property by `_id` (essential for most operations, such as "see property details").
- Find properties in a city (used in all analytics and primarily in filtered search, as discussed in the indexes section).

Choosing `city` as the sharding key, combined with a location-based partitioning strategy (where shards are grouped by geographically close cities), would penalize all operations that need to perform the first query in the list. This is because each query would need to scan all shards to determine where a specific document is stored.

On the other hand, choosing `_id` as the sharding key and using a hashing-based sharding algorithm presents the opposite problem: to find all properties in a given city, all shards must be queried since properties would be evenly distributed across all servers.

Both solutions introduce a significant downside by negatively impacting a high-volume operation (either "See property page" or "Search properties").

Our proposed solution is to slightly modify the operations so that, for all queries that need to find a property by `_id`, the `city` is also transmitted. Since the city is always known when the `_id` of a property is known, due to the way data is structured in MongoDB, this approach enables location-based sharding using `city` as the sharding key. This ensures that each shard contains a comparable number of properties, balancing the load across servers.

Seller

This collection is used for the following queries:

- Get a seller by `_id` (used by all other operations performed on this collection).
- Get a seller by email (used for login).

For this collection, we propose using `_id` as the sharding key with a hashing-based sharding algorithm to evenly distribute the load across all servers. Although the login operation would be penalized, since all servers would need to be queried to

find the email, this operation is performed only once per session and can be further optimized using cookie-based mechanisms.

Buyer

The basic operations performed on this collection are:

- Get a buyer by `_id` (used by all other operations performed on this collection).
- Get a buyer by email (used for login).

Similar to the `Seller` collection, we propose using `_id` as the sharding key with a hashing-based sharding algorithm to balance the load across all servers. Although the login operation would require querying all servers to locate a specific email, this operation is performed only once per session and can be optimized through cookie-based mechanisms.

7.2.2 Sharding on Redis

Considering that the key-value pairs used for managing reservations occupy only a few megabytes, and given Redis guidelines that recommend sharding only for datasets larger than 25GB, we have decided not to implement sharding. This decision brings several significant advantages:

- **Simplicity:** A single instance is easier to manage, configure, and monitor, thereby reducing the overall architectural complexity.
- **Lower Overhead:** The absence of partitioning eliminates the need for coordination among multiple nodes, thus reducing administrative and operational overhead.
- **Reduced Latency:** All access operations occur on the same instance, avoiding delays caused by inter-node communication.

If the data volumes grow beyond the current limits, we would consider implementing a key-based hashing mechanism to maintain rapid access times.

7.3 Virtual Machine Organization

7.3.1 Structure

The virtual machine cluster is organized as follows:

- **10.1.1.107:** Primary node for MongoDB and master for Redis.
- **10.1.1.84:** Secondary node for MongoDB, slave for Redis, and web server.
- **10.1.1.87:** Secondary node for MongoDB, slave for Redis, and instance of Neo4j.

This configuration has been designed to optimally balance the load among the machines. Although the same machine is used for the MongoDB primary and the Redis master, specific strategies have been adopted to minimize the load on that node.

Tests have shown that this configuration ensures system availability even if the machine hosting both the Redis Master and the MongoDB primary fails. However, the remaining two machines experience increased latency due to the additional workload.

7.3.2 MongoDB

ReplicaSet

The ReplicaSet consists of three nodes with the following configuration:

```
{ _id: 0, host: 'mongo_machine1:10.1.1.107', priority: 5 },
{ _id: 1, host: 'mongo_machine2:10.1.1.84', priority: 2 },
{ _id: 2, host: 'mongo_machine3:10.1.1.87', priority: 1 }.
```

This configuration favors, in the event of an election, the secondary node that hosts the web server, thereby avoiding overloading the machine that manages Neo4j, which already handles computationally intensive operations.

Write Concern and Read Preference

To optimize performance:

- The **Write Concern** is set to 1, allowing for faster, leaner write operations. This choice is justified by the presence of two masters on the same machine and tolerance for data that is not updated in real time.
- The **Read Preference** is configured as `secondaryPreferred`, aligning with the requirement to tolerate some latency in data updates.

7.3.3 Redis

Replication and Redis Sentinel

Redis is distributed across all three machines using a master-slave architecture:

- The **master** handles all write operations, which are asynchronously propagated to the **slaves** that are enabled for reads.

The default Redis configurations do not include automatic master election in the event of a failure. To address this, three **sentinels** have been implemented to monitor the nodes and, in the event of a master failure, elect a new master. Although this introduces some overhead, the benefit in terms of resilience is considered more than justified.

Eviction Policy

To prevent overload due to a higher-than-expected number of reservations, an eviction policy has been activated on the Redis servers:

- `maxmemory 512mb`
- `maxmemory-policy allkeys-lfu`

The choice of the `allkeys-lfu` policy ensures that, when memory needs to be freed, the least frequently used keys are removed first, indirectly penalizing users who use the reservation service less frequently.

Persistence

Despite the volatile nature of reservations, it is crucial to ensure data persistence in case of node failures. Therefore, periodic saving using RDB (Redis Database) has been enabled:

- `save 900 1` # If at least 1 key changes within 15 minutes, a snapshot is taken.
- `save 300 10` # If at least 10 keys change within 5 minutes, a snapshot is taken.
- `save 60 10000` # If at least 10,000 keys change within 1 minute, a snapshot is taken.

7.3.4 Neo4j

Due to the inherently connected nature of the data and the critical importance of relationships, Neo4j is implemented as a single instance within the cluster. Its architecture is not suitable for sharding. Moreover, since replication is not supported in the Community Edition, a single instance was chosen despite the fact that a replicated configuration could eliminate a single point of failure, despite the increased complexity.

7.4 Considerations on the CAP Theorem

In line with the non-functional requirements, the architecture is oriented towards the AP (Availability and Partition Tolerance) model, thereby allowing for eventual consistency:

- **Internal Eventual Consistency:** Within each database, consistency is gradually achieved through replication.
- **Eventual Consistency Across Databases:** The presence of multiple copies (e.g., between MongoDB and Redis) allows the system to tolerate data that is not updated immediately, in favor of higher availability.

To manage operations that involve multiple databases or in the event of a failure, several strategies have been implemented:

- **Rollback:** In case of failed operations, a rollback is attempted, and if it fails, a log is recorded for technical intervention.
- **Failure Tolerance and Retry:** Non-critical operations can be retried asynchronously to ensure data consistency.
- **Immediate Update vs. Outdated Data:** For some reservations, an immediate update is preferred, while for others deemed non-essential, a slight latency (up to one week) is tolerated in order to achieve consistency.

This approach ensures that, over time, the application reaches a consistent state while maintaining high availability and resilience in the presence of partitions or failures.

Chapter 8

Future Works

8.1 Potential Enhancements for Further Improving the Project

- Implement a penalty system for users who register for open house events but do not attend.
- Develop a dedicated interface that functions as a back office for the technical staff and an hypothetical superuser; this platform interacts with the back-end, which would expose specific extra endpoints to act directly on the databases with CRUD operations (currently, they can use the functions in `db_entities`, but there are no dedicated APIs available for them).
- Introduce a cron job to automatically remove old sold properties from the agency's listings.
- Provide global (multi-agency) statistics to enhance market analysis.

Appendix A

Liveability Scoring System

A.1 Properties

This algorithm is designed to calculate a liveability score for properties based on nearby Points of Interest (POI). The score reflects the quality of a property's location, taking into account the number, type, and distance of POIs. It uses an exponential function to balance the contributions from each POI, providing a meaningful and interpretable score.

A.2 Algorithm Description

The algorithm relies on the following components:

A.2.1 Input Data

- **POI**: A set of points of interest near the property, which are associated with a weight w_i.
- $\mathbf{d_i}$: The distance from the property to the point of interest (in meters or kilometres).
- **N**: The number of POIs connected to the property.

A.2.2 POI Weight

Each POI type is assigned a weight w_i that represents its importance for liveability.

POI	**Weight** (w)
amenity: hospital	0.5
amenity: school	0.2
leisure: park	0.3
amenity: police	0.2
shop: supermarket	0.3
amenity: kindergarten	0.1
industrial: factory	-0.2
landuse: landfill	-0.3
amenity: prison	-0.4
amenity: graveyard	-0.1

A.2.3 Distance Weighting Function

The relevance of each POI decreases with distance, using an exponentially decaying function:

$$f(d_i) = e^{-d_i} \tag{A.1}$$

.

A.2.4 Total Score Calculation

For each POI, its contribution is calculated as:

$$C_i = w_i \cdot f(d_i) \tag{A.2}$$

Summing all contributions yields:

$$x = \sum_{i=1}^{N} C_i \tag{A.3}$$

A.2.5 Normalization with Exponential Function

The final liveability score is derived using:

$$S = 100 \cdot (1 - e^{-x}) \tag{A.4}$$

This ensures that:

- the score approaches 100 for high x values (high liveability).
- the score approaches 0 for low x values (low liveability). Negative POIs are rare compared to positive ones, but the score could theoretically also be negative.

Appendix B

Test

Since the use of the RESTful APIs is well documented in Swagger with examples and response models, below we report only the principal use cases of the application for the various user types along with the real tests performed. The total request execution time (measured using POSTMAN) is also indicated. For the following examples, buyer account with email `jasonclements@buyer.com` with password `testtest`, seller account with email `compass@seller.com` and password `testtest` were used.

B.1 Simulated Property Search by a Guest/Buyer

Filtered Search
Request:

- **Type:** POST
- **Path:** `/guest/properties_on_sale/search`
- **Parameters:** { page=1, page_size=10 }
- **Body:**

```
{
  "city": "New York",
  "neighbourhood": "Diamond District",
  "max_price": 2000000,
  "min_area": 200,
  "min_bed_number": 3,
  "min_bath_number": 2
}

```

Response:

- **Response Time:** 50ms
- **Code:** 200
- **Body:** A list of 6 `PropertyOnSale` objects containing only the summary information about the properties.

See Property Page
Request:

- **Type:** GET
- **Path:** `/guest/property_on_sale/67acbd92ef14e7015bcdaf39`
- **Parameters:** `property_on_sale_id=67acbd92ef14e7015bcdaf39` (one of the properties in the previous list)

Response:

- **Response Time:** 46ms
- **Code:** 200
- **Body:** A `PropertyOnSale` object containing all the information present in the database for the queried property.

See Property and Near Properties on the Map
Request:

- **Type:** GET
- **Path:** `/guest/map/properties_near_property`
- **Parameters:** `property_on_sale_id=67acbd92ef14e7015bcdaf39`

Response:

- **Response Time:** 84ms
- **Code:** 200
- **Body:** A list of `PropertyOnSaleNeo4J` objects containing summary information about properties near the queried property, including their coordinates and livability score.

See Near POIs
Request:

- **Type:** GET
- **Path:** `/guest/map/pois_near_property`

- **Parameters:** `property_on_sale_id=67acbd92ef14e7015bcdaf39`

Response:

- **Response Time:** 66ms
- **Code:** 200
- **Body:** A list of `POI` objects representing the points of interest near the queried property.

View Information about City & Neighbourhood
Request:

- **Type:** GET
- **Path:** `/guest/map/city_and_neighborhood`
- **Parameters:** `property_on_sale_id=67acbd92ef14e7015bcdaf39`

Response:

- **Response Time:** 55ms
- **Code:** 200
- **Body:** An object of type `City` and an object of type `Neighbourhood` containing summary information and the location of the city and neighbourhood where the property is located.

Add to Favourite List (Buyer Only)
Request:

- **Type:** POST
- **Path:** `/buyer/favourite`
- **Body:**


```
{
  "property_on_sale_id": "67acbd92ef14e7015bcdaf39",
  "thumbnail": "https://photos.zillowstatic.com/fp/0d717658d7b
   85c223735428e259c8878-p_e.jpg",
  "address": "100 W 57th St UNIT 4JK, Manhattan, NY 10019",
  "price": 329000,
  "area": 2200
}

```

All the information in the request body is already known to the client because the user previously executed the "See Property Page" operation.

Response:

- **Response Time:** 42ms
- **Code:** 200
- **Body:** A success message confirming the operation.

B.2 Market Analytics Usage

Average Price per Square Meter by Neighbourhood for a Given City (with optional filtering by property type)
Request:

- **Type:** POST
- **Path:** `/registered-user/analytics_1`
- **Body:**

```
{
  "city": "New York",
  "type": "Condo"
}

```

Response:

- **Response Time:** 74ms
- **Code:** 200
- **Body:** A success message with a list of New York neighbourhoods showing the average price per square meter for properties of type "Condo", sorted in ascending order.

Average Selling Time in Days per Neighbourhood for a Given City and Time Period (Seller Only)
Request:

- **Type:** POST
- **Path:** `/registered-user/analytics_2`
- **Body:**

```
{
  "city": "New York",
  "start_date": "2025-02-01",
  "end_date": "2025-03-31"
}

```

Response:

- **Response Time:** 48ms
- **Code:** 200
- **Body:** A success message with a list of neighbourhoods where the seller has sold at least one property, including the number of properties sold and the average selling time (in days) during the specified period.

B.3 Daily Operations by a Seller

View Properties for Sale
Request:

- **Type:** GET
- **Path:** `/seller/properties_on_sale`
- **Parameters:** { city=Los Angeles, neighbourhood=Silver Triangle }

Response:

- **Response Time:** 50ms
- **Code:** 200
- **Body:** A list of the seller's current properties for sale, including summary information.

View Today's Open House Events for a Given City
Request:

- **Type:** GET
- **Path:** `/seller/current_open_house_events`
- **Parameters:** { city=Los Angeles}

Response:

- **Response Time:** 57ms

- **Code:** 200
- **Body:** A list of the seller's today open house event with time and address of the properties involved.

View Reservations for a Property
Request:

- **Type:** GET
- **Path:** `/seller/property_on_sale/reservations`
- **Parameters:** `property_on_sale_id=67acbd92ef14e7015bcdaf62`

Response:

- **Response Time:** 52ms
- **Code:** 200
- **Body:** A list of people registered for the next open house event of the house.

B.4 Less Frequent Operations by a Seller

Sell a Property
Request:

- **Type:** POST
- **Path:** `/seller/property_on_sale`
- **Body:** { `property_to_sell_id`= *(ID)* }

Response:

- **Response Time:** 492ms
- **Code:** 200
- **Body:** A success message indicating that the property was sold.

Add a New Property
Request:

- **Type:** POST
- **Path:** `/seller/property_on_sale`
- **Body:**

```
{
```

```
  "city": "New York",
  "neighbourhood": "Bronx",
  "address": "12 Park Place",
  "price": 270000,
  "thumbnail": "http://example.com/photo.jpg",
  "type": "condo",
  "area": 100,
  "bed_number": 3,
  "bath_number": 2,
  "description": "Beautiful home",
  "photos": [
    "http://example.com/photo1.jpg"
  ]
}

```

Response:

- **Response Time:** 3s
- **Code:** 200
- **Body:** A success message along with the `_id` of the newly created property.

Note that the execution time for this query is significantly longer than others due to the insertion of the property into Neo4j. This is justified in the section on critical operations.