Developing a Spatial Pattern Matching Application

Chomondozlis Paschalis

Dissertation Report

Supervisor: N. Mamoulis

Ioannina, October 2020

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Acknowledgements

I would like to thank first and foremost my supervisor, Nikolaos Mamoulis, for the invaluable hours of guidance and patience he has given me. He was always quick and kind to assist me whenever I requested it.

Lastly, I would like to extend my very warm gratitude to my family and friends whom without their support, I would not have been able to make it this far.

October, 2020

Paschalis Chomondozlis

Περίληψη στα ελληνικά

Σε αυτήν εργασία αναπτύσσουμε ένα προγραμματιστικό εργαλείο πλήρης στοίβας το οποίο προορίζεται για την λύση προβλήματος εύρεσης ικανοποιητικών στοιχείων σε χωρικά μοτίβα και προορίζεται για χρήση στον χώρο της πώλησης και ενοικίασης βιομηχανικών ακινήτων. Θα αναλύσουμε το πρόβλημα του πραγματικού κόσμου που προσπαθεί να λύσει καθώς και όλα τα βήματα και οι τεχνικές που ακολουθήθηκαν κατά την ανάπτυξη αυτού του εργαλείου όπως τον αλγόριθμο εύρεσης εκείνων των ικανοποιητικών στοιχείων. Συγκεκριμένα, θα εξετάσουμε την συλλογή και αποθήκευση δεδομένων κρίσιμα για την απρόσκοπτη λειτουργία της εφαρμογής μας, την ανάπτυξη ενός συστήματος λογισμικού API το οποίο θα μας προσφέρει χρήσιμες υπηρεσίες προς την εκπόνηση του βασικού στόχου και τέλος την ανάπτυξη μιας διεπαφής χρήστη που θα είναι συμβατή με τα σύγχρονα στάνταρ και θα εξυπηρετεί τον χρήστη εύκολα και γρήγορα. Η εφαρμογή αυτή προορίζεται να τοποθετηθεί σε ένα διακομιστή και να είναι προσβάσιμη από το ευρύ κοινό με τη χρήση ενός προγράμματος περιήγησης στο Ίντερνετ. Θα χρησιμοποιήσουμε εξωτερικές υπηρεσίες της Google όπως το Google Maps API και το Firestore Datastore για βάση δεδομένων. Θα εξετάσουμε τρόπους ελέγχου των ορίων της εφαρμογής καθώς και θα παρουσιάσουμε αποτελέσματα αυτών προσομοιώνοντας την χρήση της από πολλαπλούς χρήστες ταυτόχρονα. Τέλος, θα προτείνουμε πιθανές επεκτάσεις της που μπορούν να υλοποιηθούν στο μέλλον.

Λέξεις Κλειδιά: αναζήτηση, βιομηχανικά, ακίνητα, χωρικά, μοτίβα, πλήρης, στοίβα, διεπαφή, διακομιστής, βάση, δεδομένων, συλλογή

Abstract

In this project we will develop a full-stack application which aims at solving a problem of spatial pattern satisfaction. It is intended to be used by the real-estate market as it provides a way for finding available commercial properties. We will analyze the real-life problem that it strives to solve as well as all the tools and techniques we used for developing this application. In detail, we will show how we managed to collect and store all the data needed for the seamless operation of the application, how we developed a backend API capable of offering services aimed at helping us build the tool and the creation of a user interface that's robust and up to modern standards to serve the client fast and easily. This application will be deployed on a web server and be accessible through a web browser by the public. We will integrate technologies such as the Google Maps API and the Firestore Datastore which are provided by Google to assist us in crucial operations. Upon deployment, we lay out ways that we stress test our application to find its limits and capabilities and we present the results of those tests along with the necessary commentary. In conclusion of this paper, we analyze possible extensions that could be implemented in the future.

Keywords: search, real-estate, commercial, property, spatial, patterns, full-stack, application, Spring Boot, React, Java, JavaScript, database, Firestore, engine, dataset, server, backend, frontend

Table of Contents

Chapter 1. Introduction

1.1 Subject of the dissertation

Logistics and supply chains have always played a key role in building an efficient and up-to-date business, capable of catering to the ever-growing market demands. Modern day enterprises consist of many stores that require constant resupplying from warehousing units located in strategic places that offer fast access to those stores as well as vast storing capabilities. That's where our application comes into play, offering a way to locate available commercial listings around already established stores.

The dissertation's subject as a whole, including the methodologies and experimentation conducted, all fall within the researching fields of software development, information retrieval as well as geospatial data processing with complex data structures.

In a macroscopic view, the project's overall goal is to develop a full stack web application that is able to accept and process custom user queries and return back results based on said query. Now, in a more in-depth analysis, these queries serve the purpose of searching for real-estate, particularly commercial real-estate, that the application holds in its database by querying *around* already established facilities that the user owns. It is intended to function as a tool for searching warehousing units that are close to the user's retail stores and provide a solid resupplying chain.

Diving in the terms mentioned above, *commercial real-estate* is defined as properties intended to generate profit either from capital gain or rental income. In continuation, *commercial buildings* are defined as buildings intended for commercial use. Such real estate exists in the form of many categories and types namely offices, warehouses, retail buildings, hotels.

From our search, we concluded that no other real-estate searching tool offers this kind of functionality or targets this particular subcase of searching. Meanwhile, the

searching for warehousing units and the need for efficient resupplying chains grows ever more rapidly making the tool that we develop a serious contender in the market.

In conjunction with the above, we intend to provide a general direction and insight into developing a full stack web application by presenting our design choices, methodology, testing and evaluation of the end result. In particular, we create a fully-fledged, extensible and, above all, maintainable API that is used along with a robust and user-friendly client interface that is deployed as a web application on Amazon Web Services.

1.2 Report Structure

We begin by stating the requirements and aims, provide a general overview of the ways we intend to meet them in a formal Software Requirement Specification document as well as present the data model that will be utilized by the application.

We move on to describing the implementation details and the internal structure of the application. The architectural design is demonstrated along with the data structures that make up this application before we delve deeper into analyzing the specific parts on the internal structure. We mention ways of code maintenance and we close by listing a walkthrough of the app.

On the next chapter takes place the experimentation based on the application. We test its capabilities and scalability through stress testing it methodically and presenting the results.

We close with a few notes on the whole project alongside possible extensions.

Chapter 2. Software Requirements Specification

2.1 Purpose

The purpose of this section is to present a detailed description of the project by laying out the aims and features of the system, its interfaces and capabilities as well as its constraints under which it must operate. This part is intended for developers of the system and stakeholders alike.

The main goal of this project is to provide a completely customizable solution for users in search of *commercial listings*. We believe that an easy-to-use and up-to-date application that takes a more targeted approach on a particular subset of uses of real-estate search engines has a place within the market.

The intended target audience are primarily people that are currently running a business or are in a management position in one and look expanding, franchising or improve their throughput. What this application offers is centered around logistics and resupplying with the ability to expand into other areas with ease and speed. At the same time, up-and-coming businesses can also be considered a potential user of the application as it provides a searching ground for new retail stores of all price ranges and different selling types.

Finally, the scope of the project is to develop an application that will be self-maintainable by keeping its data up-to-date, be easily expandable and robust but at the same time keep its size and capabilities in feasible-to-implement levels. The data that will be comprising our app will be of the *contiguous* United States of America (excluding

Alaska & Hawaii). This is due to the small amount of data in these areas to make it worthwhile to deploy there too.

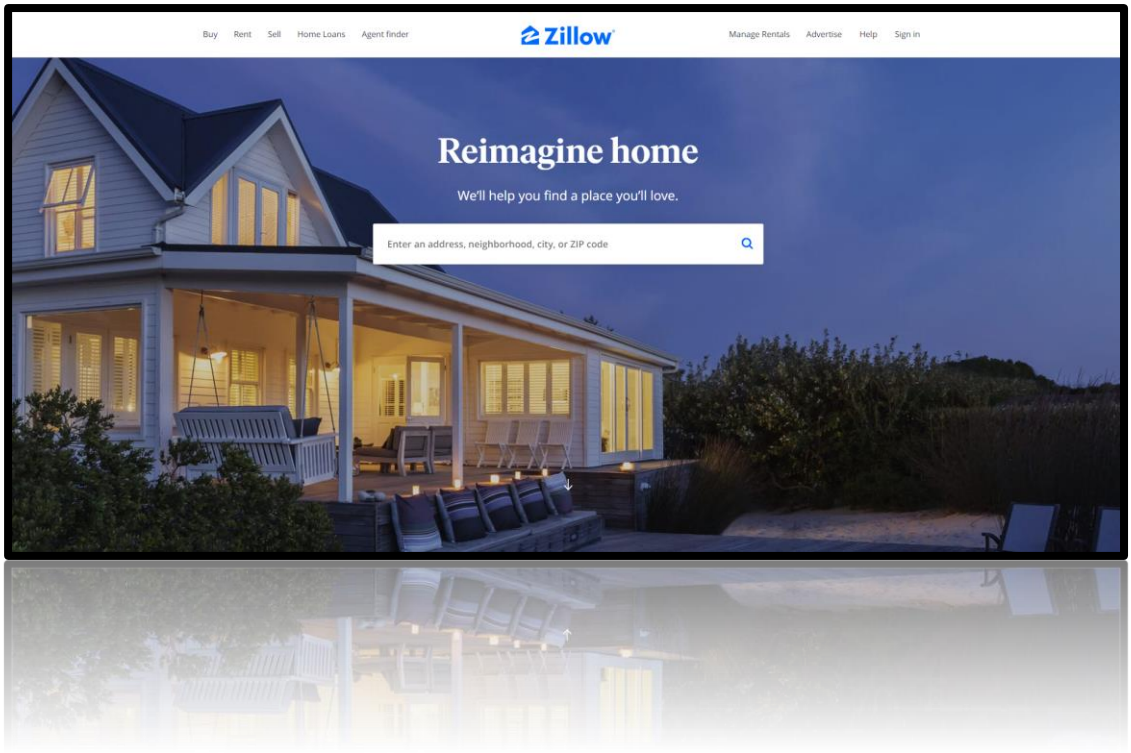
2.2 Definitions and Acronyms

- **Geopoint:** A point on a map described by a pair of coordinates, latitude and longitude.
- **Point(s) of Interest (PoI):** All the geopoints that the user of the application might be interested in. These points can be considered to represent already established retail stores, resupplying stations, warehousing units or whatever else the user wants them to be. The application makes no assumptions and leaves it entirely on the user to customize each point to their liking.
- **User:** A customer or a system administrator. Generally, someone who utilizes the system interface.
- **Stakeholder:** Someone with an interest in the project without being one of the developers.
- **Database:** Collection of all system's information.
- **Google Firestore** or **Firestore:** The service providing the database supporting the system.
- **Firebase Authentication** or **Firebase Auth:** The service providing user authentication for the system.
- **Input field:** An interface for direct user input. Usually within forms.

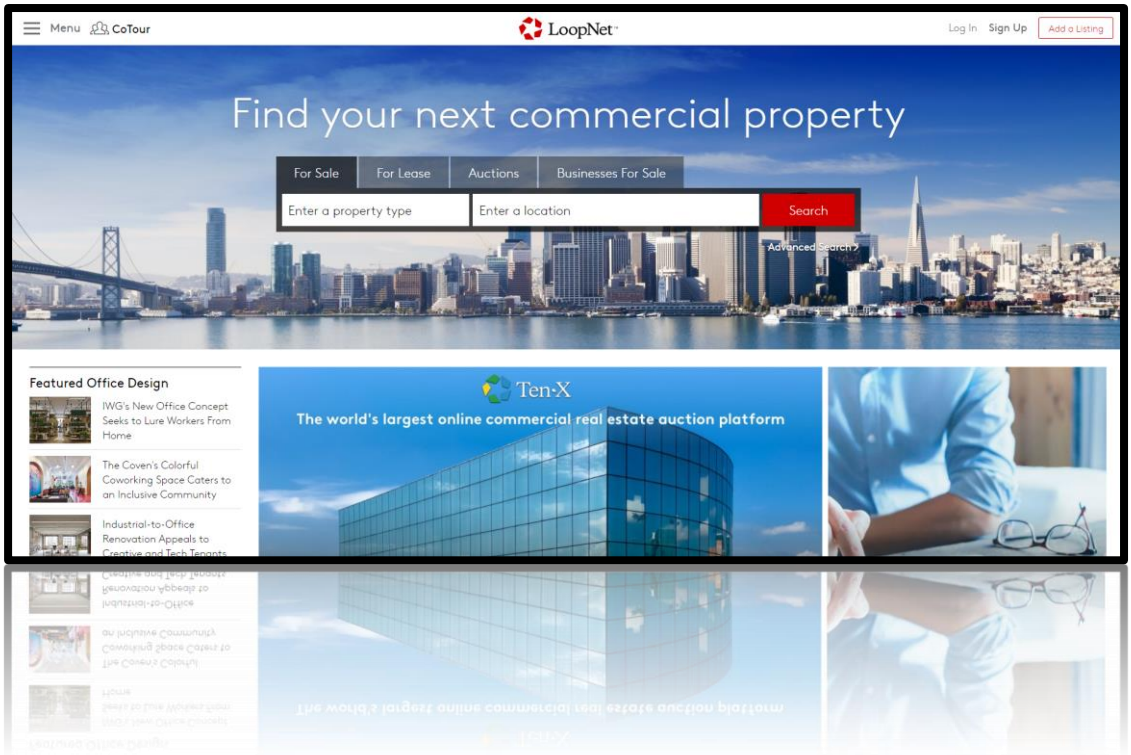
2.3 Similar technologies and applications

The idea of creating an application capable of searching for commercial listings and especially for real-estate in general is by no means groundbreaking. They are abundant and have existed for many years in a tightly packed market. Some examples of sites that employ this kind of technologies are:

- zillow.com


Buy
Rent
Sell
Home Loans
Agent finder
Zillow
Manage Rentals
Advertise
Help
Sign in
Reimagine home
We'll help you find a place you'll love.
Enter an address, neighborhood, city, or ZIP code

- loopnet.com


Menu
CoTour
LoopNet
Log In
Sign Up
Add a Listing
Find your next commercial property
For Sale
For Lease
Auctions
Businesses For Sale
Enter a property type
Enter a location
Search
Advanced Search
Featured Office Design
IWG's New Office Concept Seeks to Lure Workers From Home
The Coven's Colorful Coworking Space Caters to an Inclusive Community
Industrial-to-Office Renovation Appeals to Creative and Tech Tenants
Ten-X
The world's largest online commercial real estate auction platform

These platforms are a staple in the real-estate search engine market with more that 5 million unique visitors each month per platform and over 500.000 listings in their database. While we will draw some inspiration from strategies that they employ in the form of basic operations such as address searching and mapping search results on a map for a more comprehensive understanding of them, our system will include many unique features that, in turn, make it unique by itself.

2.4 Expected user characteristics

We expect that the average user of our application is knowledgeable in browsing the Internet through a browser or a similar tool and has a general understanding of common practices and indications for different operations/ functions. In more detail, we assume that knowledge of opening dropdown menus, switching in-app tabs and locating outlined buttons.

2.5 Requirement Analysis

In this section we will describe the requirements that our application must meet. We will present them in the form of UML use cases based on the following types of requirements:

- **Functional Requirements:** These types of requirements cover the functionality of the product and are also known as features or capabilities.
- **Non-Functional Requirements:** These are the requirements that describe a constraint on the provided services. The constraints can be of timing type, development process related, standards and many more.

The main actor in these use cases is the customer or a system administrator.

2.5.1 Functional Requirements

2.5.1.1 Use Cases

ID:	**UC1**
Title:	LogInUser
Description:	The actor is able to log in to the system.
Primary Actor:	User
Preconditions:	1. The actor is not logged in the system. 2. The actor has a registered account in the system.
Postconditions:	1. The actor is logged in the system. 2. The actor has access to the system's functions.
Main Success Scenario:	1. The use case begins when the user clicks the "Login" button. 2. The system presents a form with the login credentials. 3. The actor enters their login credentials. 4. The system validates the actor's credentials and logs them into the system. 5. The system redirects the actor on the landing page and the use case ends.
Extensions:	1. Incorrect credentials 1.1. The system cannot identify the actor. 1.2. A message is displayed stating that they have entered the wrong credentials. 1.3. The actor is prompted to enter their credentials again. 1.4. The use case continues at step 4.
Frequency of Use:	Rarely.
Status:	Completed.

ID:	**UC2**
Title:	LogOutUser
Description:	The actor is able to log out of the system.
Primary Actor:	User
Preconditions:	1. The actor is logged in the system.
Postconditions:	1. The actor is logged out of the system. 2. The actor does not have access to the system's functions.
Main Success Scenario:	1. The use case begins when the actor clicks the "Logout" button. 2. The system logs the actor out of the system. 3. The system redirects the actor unto the landing page and the use case ends.
Extensions:	The are no exceptions for this use case.
Frequency of Use:	Rarely.
Status:	Completed.

ID:	**UC3**
Title:	SignUpUser
Description:	The actor is able to create an account in the system.
Primary Actor:	User
Preconditions:	1. The actor is not logged in the system. 2. The actor does not have an account registered.
Postconditions:	1. The actor is logged in the system. 2. The actor has access to the system's functions.
Main Success Scenario:	1. The use case begins when the actor clicks the "Sign Up" button. 2. The system redirects the actor unto the sign-up form. 3. The actor completes the form with his own credentials. 4. The system validates the given credentials and generates a request for creating a new user. 5. The system receives a confirmation of the process and informs the actor. 6. The system logs in the actor and redirects to the landing page. 7. The use case ends.
Extensions:	1. The system receives an error during account creation: 1.1. The actor is notified. 1.2. The system aborts the operation and the use case ends. 2. The system detects invalid credentials: 2.1. The system informs the actor of the reason and prompts them to try again. 2.2. The use case continues from Step 4.
Frequency of Use:	Rarely.
Status:	Completed.

ID:	**UC4**
Title:	ViewApplicationForm
Description:	The actor is able to view a form of the application.
Primary Actor:	User
Preconditions:	1. The actor is logged in. 2. The actor is viewing the landing page.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor clicks the “Start” button on the landing page. 2. The system redirects the actor unto the main application forms (“Map Form”) and the use case ends.
Extensions:	1. Actor is not logged in. 1.1. The system redirects the actor unto the login page. 1.2. include(LogInUser) 1.3. The use case ends.
Frequency of Use:	Rarely.
Status:	Completed.

ID:	UC5
Title:	NavigateApplicationForms
Description:	The actor is able to freely navigate the forms offered by the application.
Primary Actor:	User
Preconditions:	1. The actor is logged in. 2. The actor is browsing the "/main" subdirectory of the application.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor is presented an application form that exists inside the "/main" application path. 2. The actor presses a tab icon of their own choice from the left sidebar. 3. The system highlights and presents the corresponding tab's form and the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	UC6
Title:	ViewMap
Description:	The actor is able to view the map.
Primary Actor:	User
Preconditions:	1. The system renders the Map Form.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor selects the "MAP" button inside the Map Form. 2. The system presents the map along with a helpful toolbar to the actor and the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC7**
Title:	SelectMapToolbarFunction
Description:	The actor is able to freely navigate the map's toolbar tabs.
Primary Actor:	User
Preconditions:	1. The actor is logged in. 2. The actor is browsing the "/main" subdirectory of the application. 3. The system is rendering the map toolbar.
Postconditions:	1. The actor has enabled a map functionality.
Main Success Scenario:	1. The use case begins when the actor is viewing the map within the Map Form tab contents. 2. The system has by default selected the first ("Compass') tab icon. 3. The actor hovers over a tab icon. 4. The system displays a helper text above the icon explaining its functionality. 5. The actor selects a tab icon on the map toolbar. 6. The system de-highlights any previous selection, disables any of its functionality and highlights the tab icon. 7. If the actor selects the first ("Compass") icon: 7.1. The system enables map panning. 8. If the actor selects the second ("Marker") icon: 8.1. The system enables map panning. 8.2. The system enables the ability for markers to be placed. 9. If the actor selects the third ("Search") icon: 9.1. The system displays an input box above the toolbar. 10. The use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC8**
Title:	PanFreelyOnMap
Description:	The actor is able to freely pan over the map.
Primary Actor:	User
Preconditions:	1. The system is rendering the Google map.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor presses the left click button while moving the mouse cursor over the map. 2. The system accepts the input given and moves the map accordingly. 3. The system renders a new area of the map and the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	UC9
Title:	PlacePoIOnMap
Description:	The actor is able to place custom points of interest (markers) over the map.
Primary Actor:	User
Preconditions:	1. The system is rendering the Map form. 2. The actor has selected the second ("Marker") tab icon on the map toolbar.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor double presses the left click button over a point on the map. 2. The system generates a request for placing a new marker on the map. 3. The system returns that the processing of the request was successful. 4. The system places a marker over the point on the map and the use case ends.
Extensions:	1. The system returns unsuccessful request processing. 1.1. The actor is notified that the request failed. 1.2. The use case ends.
Frequency of Use:	Often.
Status:	Completed.

ID:	UC10
Title:	ViewPoI
Description:	The actor is able to view all their points of interest.
Primary Actor:	User
Preconditions:	1. The system is rendering the Map form.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor selects the "POINTS OF INTEREST" button inside the Map Form. 2. The system presents the table containing all the actor's points of interest and the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC11**
Title:	SelectPoI
Description:	The actor is able to select any number of points of interest.
Primary Actor:	User
Preconditions:	1. The system is rendering the PoI table
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor selects a row on the PoI table. 2. The system highlights the selected row and updates the table header with the total number of points of interest currently selected. 3. While the actor selects rows: 3.1. The system highlights the selected rows and update the table header. 4. The use case ends
Extensions:	
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC12**
Title:	DeletePoI
Description:	The actor is able to delete any number of points of interest.
Primary Actor:	User
Preconditions:	1. The system is rendering the PoI table
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor presses the “Delete” icon button on the table header. 2. The system generates a request for deleting the selected PoI. 3. The system deletes locally the PoI. 4. The system receives a successful confirmation of the processing of the request, notifies the actor and the use case ends
Extensions:	1. The system receives a notification that the processing of the request was unsuccessful: 1.1. The system notifies the actor. 1.2. The system rolls back the local changes and the use case ends.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC13**
Title:	EditPoI
Description:	The actor is able to edit the attributes of a point of interest.
Primary Actor:	User
Preconditions:	1. The actor is viewing the PoI table.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor selects the "Edit" button on a row on the PoI table. 2. The system presents a form to the actor filled with the attributes of the point of interest. 3. The actor edits the attributes of his choice in the input fields and presses the "Edit" button on the form. 4. The system edits the PoI. 5. The system generates a request for editing the point of interest. 6. The system receives a confirmation on the successful processing of the request, notifies the actor and the use case ends.
Extensions:	1. The system receives a message of the unsuccessful processing of the request: 1.1. The system notifies the actor. 1.2. The system rolls back the changes made to the marker and the use case ends. 2. The system prevents the actor from editing the point's of interest geographical values.
Frequency of Use:	Rarely.
Status:	Completed.

ID:	**UC14**
Title:	ViewStateBoundary
Description:	The actor is able to view the boundary of a state highlighted on the map.
Primary Actor:	User
Preconditions:	1. The actor is viewing the map.
Postconditions:	1. The boundary of the selected state is outlined on the map.
Main Success Scenario:	1. The use case begins when the actor selects a state from a list of states located on the map toolbar. 2. The system highlights the boundary of the selected state on the map and adjust the viewport so that it is entirely visible. 3. The system updates the selected state on the toolbar and the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC15**
Title:	SearchAddress
Description:	The actor is able to query the map with an address to be searched.
Primary Actor:	User
Preconditions:	1. The actor is viewing the map. 2. The actor has selected the “Search” icon on the toolbar.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor types inside the search input field presented by the system. 2. The actor presses the search button. 3. The system performs a search based on the actor input. 4. If the search was successful: 4.1. The system informs the actor. 4.2. The system centers the viewport of the map above the result. 4.3. The system places a marker above the result on the map. 4.4. The system presents a panel containing the address of the result and the use case ends. 5. If the search was unsuccessful: 5.1. The actor is notified and the use case ends.
Extensions:	1. The actor may abandon the search at any time.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC16**
Title:	AddSearchResultMarker
Description:	The actor is able to add the result of the search to their list of markers.
Primary Actor:	User
Preconditions:	1. The actor has queried the system with an address and has been presented a result.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor searches for an address. 2. include (SearchAddress) 3. If the system found a result: 3.1. If the actor chooses to add it: 3.1.1.The system attempts to add the search result to the actor's PoI list. 3.1.2.If the system is successful: 3.1.2.1. The system adds the result to the actor's PoI list. 3.1.2.2. The system informs the actor. 3.1.2.3. The system removes the result and the use case ends. 3.1.3.If the system is unsuccessful: 3.1.3.1. The system informs the actor. 3.1.3.2. The system removes the result and the use case ends. 3.2. If the actor chooses not to add it: 3.2.1.The system removes the result and the use case ends. 4. If the system did not find a result, the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC17**
Title:	ViewMarkerInfo
Description:	The actor is able to view the attributes of a marker on the map.
Primary Actor:	User
Preconditions:	1. The actor is viewing the map.
Postconditions:	1. The actor is presented with the attributes of the marker.
Main Success Scenario:	1. The use case begins when the actor left clicks on a marker. 2. The system centers the map viewport around the selected marker. 3. The system presents a dialog box above the marker containing all of its attributes and the use case ends.
Extensions:	1. If the actor clicks outside of a marker, the system closes the currently open dialog box.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC18**
Title:	ApplyFilters
Description:	The actor is able to apply custom filters before executing the query.
Primary Actor:	User
Preconditions:	There are no preconditions.
Postconditions:	1. The actor's filtering preferences are updated.
Main Success Scenario:	1. The use case begins when the actor selects the "Query Filtering Form" tab. 2. The actor sets the filters to his preference. 3. The system updates the actor's filtering preferences, informs the actor and the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	UC19
Title:	ApplyRanking
Description:	The actor is able to create a custom ranking equation on which the results of the query will be sorted.
Primary Actor:	User
Preconditions:	There are no preconditions.
Postconditions:	1. The actor's ranking preferences are updated.
Main Success Scenario:	1. The use case begins when the actor selects the "Query Ranking Form" tab. 2. The actor sets the ranking formula biases to his preference. 3. The system updates the actor's ranking preferences, informs the actor and the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	UC20
Title:	ExecuteQuery
Description:	The actor is able to query the application.
Primary Actor:	User
Preconditions:	There are no preconditions.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor presses the "Search Now" button. 2. The system redirects the actor to the results' page. 3. The system executes the query, returns the results and the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC21**
Title:	ViewResults
Description:	The actor is able to view the results of the query.
Primary Actor:	User
Preconditions:	There are no preconditions.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor executes the query. 2. If the system receives results: 2.1. The system presents the results on a tab, ranked based on ranking score and the use case ends. 3. If the system receives no results: 3.1. The system informs the actor and presents a helpful message for getting better results on the next query. 3.2. The use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC22**
Title:	ViewResultsOnMap
Description:	The actor is able to view the results' locations on the map.
Primary Actor:	User
Preconditions:	1. The actor has executed a search query. 2. The system has received results.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor presses the "MAP" handlebar on the results page. 2. The system presents a map to the actor, with markers indicating all the returned results and the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Often.
Status:	Completed.

ID:	**UC23**
Title:	PreventUnauthorizedAccess
Description:	The actor is not able to access application content while unauthorized.
Primary Actor:	User
Preconditions:	1. The actor is not logged in.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor attempts to access a protected application resource. 2. The system redirects the actor unto the login page and rejects the request. 3. The use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Rarely.
Status:	Completed.

ID:	**UC24**
Title:	CloseApplication
Description:	The actor is able to close the application.
Primary Actor:	User
Preconditions:	There are no preconditions.
Postconditions:	There are no postconditions.
Main Success Scenario:	1. The use case begins when the actor wants to close the application. 2. If the actor is logged in: 2.1. The system persists the authentication by storing it in local storage. 2.2. The system closes the application and the use case ends. 3. If the actor is not logged in: 3.1. The system closes the application and the use case ends.
Extensions:	There are no extensions.
Frequency of Use:	Rarely.
Status:	Completed.

2.5.2 Non-Functional Requirements

2.5.2.1 Performance

The application will be deployed on a server capable of high throughput and able to serve multiple users concurrently. While we can guarantee a sufficient speed from our end, the overall speed will also depend on the hardware and connection of the user.

2.5.2.2 Security

The authentication credentials of each user will be kept in a different database from all the other application information. The security of both databases should be of the utmost importance as a possible leak of information may be detrimental to the operational capacity of the application. We will be working alongside reputable authentication vendor *Firebase Authentication* which is backed by *Google*. No one, not even developers of the system should have access to the authentication credentials of users.

As for the application database, it should be protected from unauthorized access by defining a proper ruleset. In the same course as with the authentication, we will be utilizing a 3rd party vendor's services, *Google Firestore,* for fast and reliable cloud storage.

2.5.2.3 Logical Data Model

The data model is presented below:

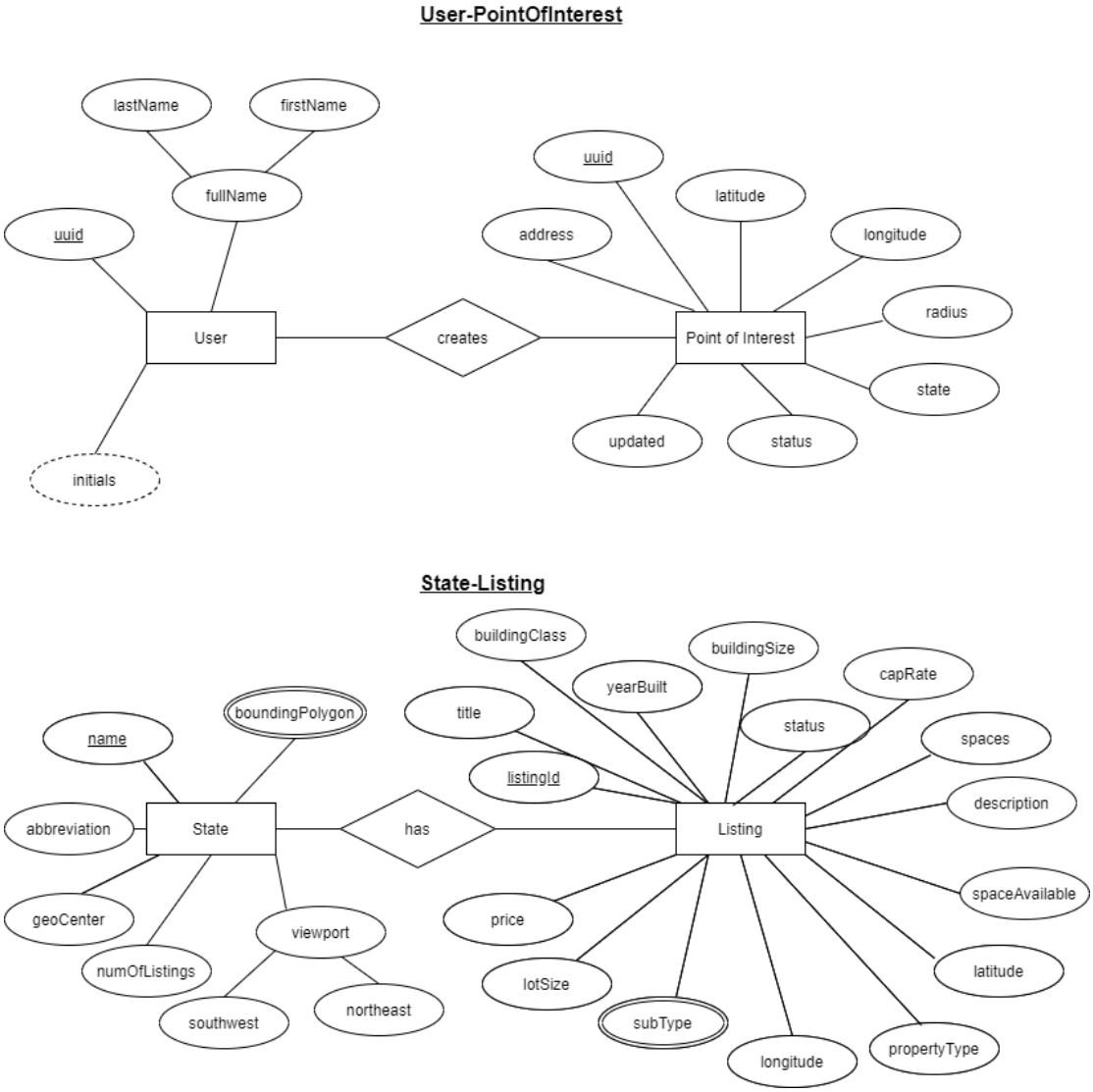


Image 1: E/R data model of the application

The entities that are shown above are:

User

Attribute	Type	Description	Additional Notes
uuid	String	A unique identifier	-
fullName	String	The user's full name	Comprises of a first and last name
firstName	String	The user's first name	-
lastName	String	The user's last name	-
initials	String	User's initials	Derived from first and last name

Table 1: User entity attributes

Point of Interest

Attribute	Type	Description	Additional Notes
uuid	String	A unique identifier	-
address	String	The address of the poi	-
latitude	double	The latitude coordinate of the poi	Values within the interval [-90, 90]
longitude	double	The longitude coordinate of the poi	Values within the interval [-180, 180]
radius	int	The radius defined around the poi	Measured in meters
state	String	The state the poi is in	-
status	Boolean	Whether the poi is active or not	-
updated	String	The date since the poi was last edited	-

Table 2: Point of Interest entity attributes

State

Attribute	Type	Description	Additional Notes
name	String	The name of the state	-
abbreviation	String	An abbreviation of the state's name	-
geoCenter	geopoint	The geographical center of the state	-
numOfListings	int	The number of listings belonging to this state	-
boundingPolygon	array<<geopoint>>	An array of geopoints defining the boundary of the state	-
viewport	map<<String, geopoint>>	Coordinates for projecting the state inside the viewport	-

Table 3: State entity attributes

Listing			
Attribute	**Type**	**Description**	**Additional Notes**
listingId	String	A unique identifier	-
title	String	The title of the listing	-
description	String	Text describing the property	-
latitude	double	The latitude coordinate of the property	-
longitude	double	The longitude coordinate of the property	-
price	double	The price of the property	Either measured in SF/Mo or a total set price
buildingClass	String	The building class of the listing	-
buildingSize	int	The building size of the listing	Measured in SF
status	String	The selling status of the listing	-
capRate	double	The capitalization rate of the property	-
yearBuilt	String	The year that the property was built	-
spaces	int	The number of spaces available within the property	-
spaceAvailable	int	The total usable space within a property	Measured in SF
lotSize	int	The total size of the property lot	Measured in SF
propertyType	String	The primary type of the property	-
subType	array<<String>>	The secondary types of the property	-

Table 4: Listing entity attributes

Chapter 3. Design & Implementation

3.1 Problem Definition

The problem of creating a web application such as this comes down to two **main** parts:

- The implementation of the application design itself and,
- The implementation of the business logic that will become part of the application structure.

In simple terms, we are trying to create a modular *husk* able to fit in any business logic, in particular, our own. While we do have to make adjustments in our architectural approach based on the business logic the core design remains the same as it would in any other application.

3.1.1 Application design

The application design will be comprised of an API functioning as the backend with a completely separate frontend interface. The business logic of our application will reside within our backend which will have, among other responsibilities, the privilege of communicating with the application's database and be in charge of granting access to that database. As for the frontend client, it will be routing all the requests for processing and data needs through the API.

3.1.2 Business Logic

We will try initially to describe the core of our application in a constructive way, starting from the simplest definition and moving on from there.

We define the following:

- **Geopoint:** A geopoint is defined as a point on the map described by a pair of coordinates in a spherical coordinate system.
- **Listing:** A listing is defined as a *commercial property* within our application's database that is also described by a pair of latitude and longitude coordinates.

In its simplest form, we define a **simple search query** as a *geopoint* with a set *radius* attributed to it. Moreover, we define the **query results** to the above query as all those listings included within the circle defined by the *geopoint* and its *radius.*

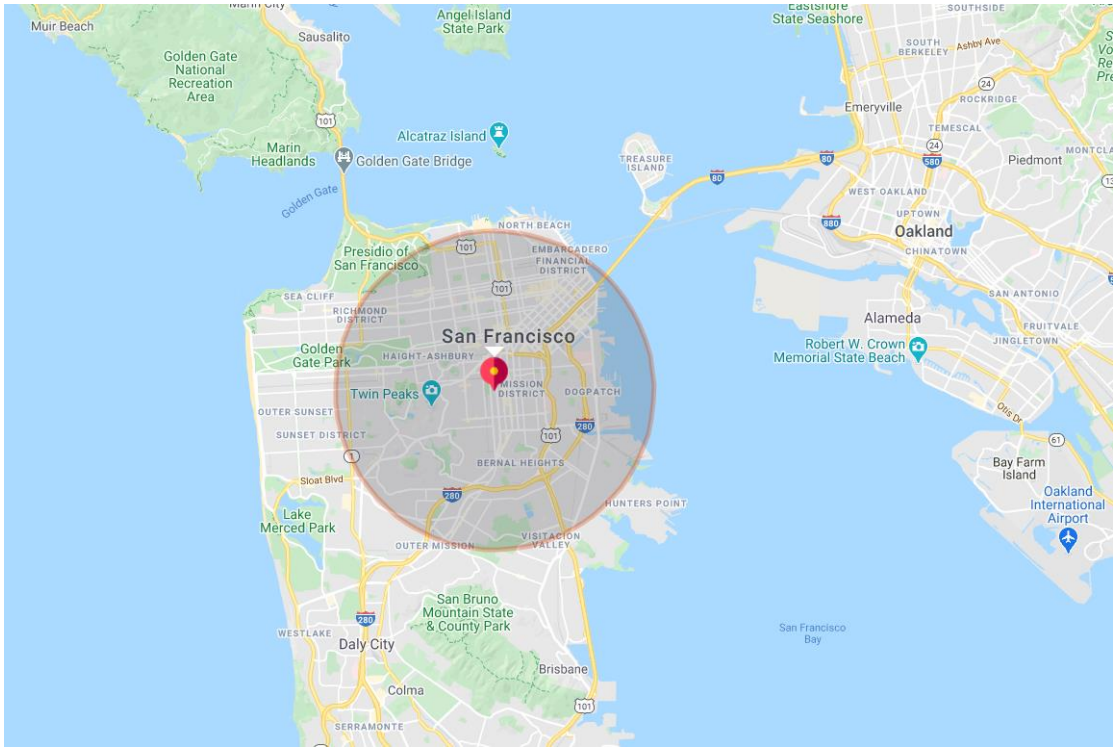


Image 2: A geopoint located in downtown San Francisco with a radius of 5km

Now, by simply combining multiple *simple search queries* we create a **search query** where the overall **query results** arise from joining the individual query results of each *simple search query*. Or even better, the **query results** are located within the intersection of all the circles defined for each *geopoint*.

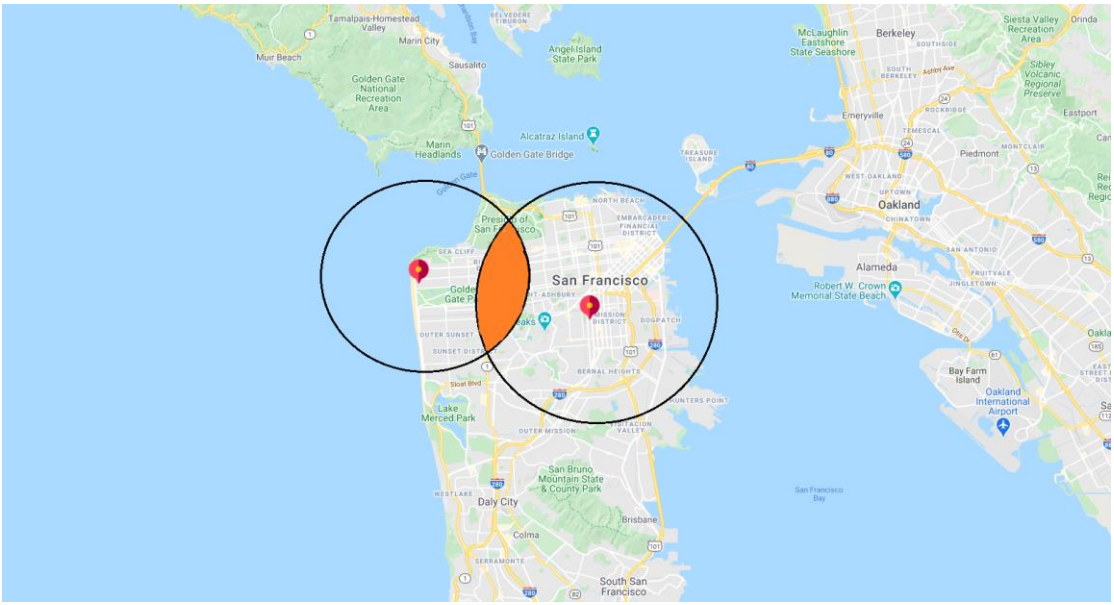


Image 3: A search query of two geopoints with their intersection highlighted

Further on we will describe the way the application handles these queries internally. But before that we need to take a deep dive into how our application is structured.

3.2 Application Architecture

Our application's architecture can be visualized using the following diagram:

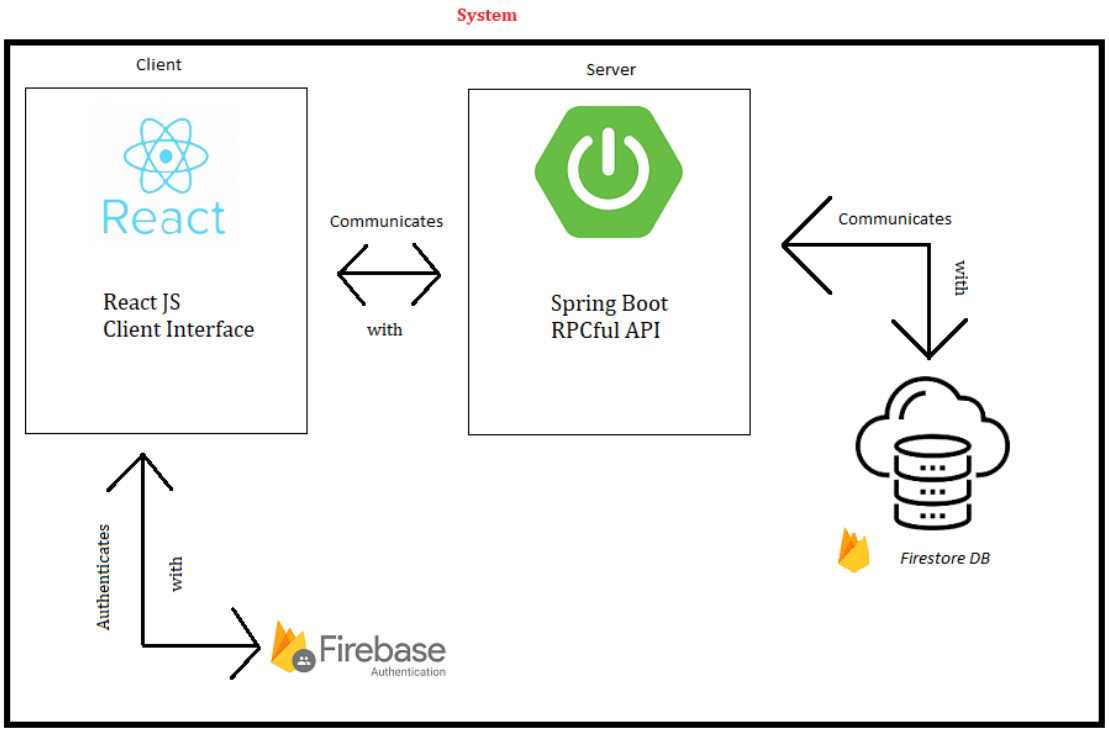


Image 4: Architecture of Warehouseify

From an eagle-eye's view our application looks like the above diagram. We can identify the following parts with their respective development platforms:

- ***Google Firestore* Database:** A cloud-based NoSQL database.
- ***Spring Boot* API:** An API interface containing all of our business logic built using the Spring Boot framework.
- ***React JS* Client Interface:** A client interface developed using the React JavaScript framework.
- ***Google Firebase Authentication*:** Authentication provider.

3.2.1 Development Tools

In this section we will try to summarise some of the benefits and utilities that the aforementioned development tools offer.

3.2.1.1 Spring Boot

The *Spring Boot* framework is an open-source Java-based framework developed by *Pivotal Team* used to create production-grade *microservices* with minimum configuration effort and are able to run as is. They offer reactivity, scalability, easy deployment and container compatibility while significantly reducing production time. *Spring boot* is recommended for responsive web application, such as this, connected to any data store.

3.2.1.2 React JS

React JS is an open-source JavaScript framework developed by *Facebook*. It streamlines trivial processes and introduces a component-based architecture for fast rendering and minimum code duplication. It heavily relies upon composition rather than inheritance.

3.2.1.3 Google Firestore

Google Firestore is a cloud-based NoSQL database maintained by *Google* that utilizes data structures known as *documents* and *collections*. Its reactivity allows for multiple concurrent non-blocking I/O connections without compromising scalability.

3.2.1.4 Google Firebase Authentication

Similarly with *Firestore*, it's maintained by *Google* and offers an authentication service with quick, out-of-the-box configuration and a multitude of sign-in methods. It also comes with the reliability of *Google* for top-of-the-line critical data security.

3.2.2 Database

Our application's database will hold data that pertain to a number of different uses which range from user to configuration data. The reason we have elected to go with a cloud-based NoSQL database, such as Firestore, is the way we want to structure and retrieve data from the database. The basic structure of a Firestore database looks like this:

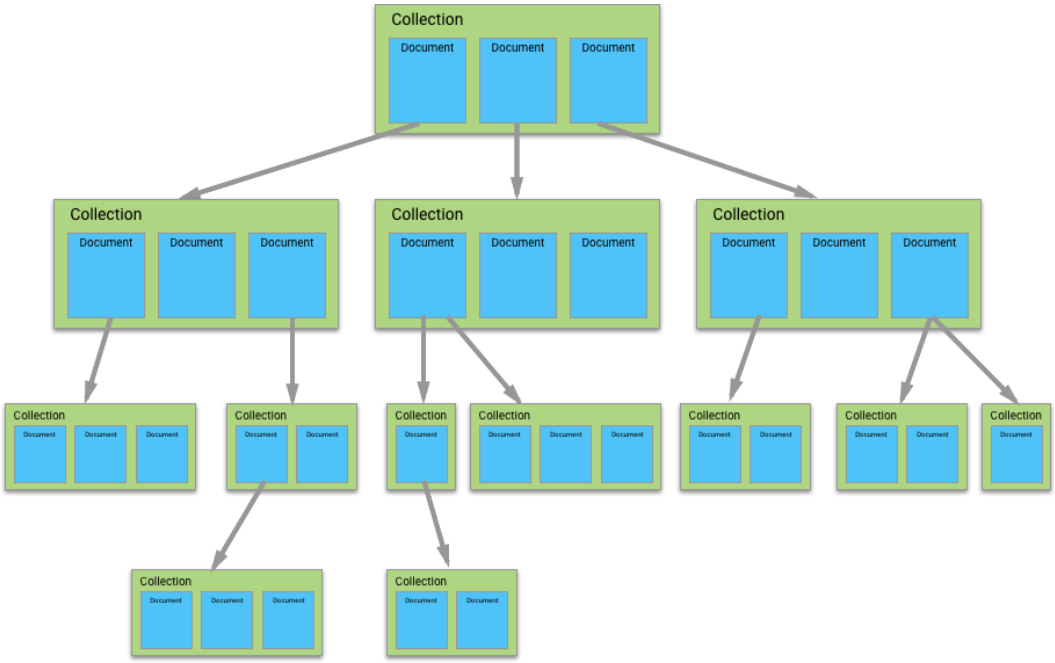


Image 5: Firestore database structure

It is organized in *collections* where each contains a number of *documents* as well as other nested collections. A *document* is the most basic type in a Firestore database and it is just a JSON object holding on a variable number of attributes. Each *document* within the same collection will have a unique ID (UUID) from which it can be queried. We can imagine that each document is a POJO, a python dictionary or any other data structure similar to those. Adapting the E/R models described in [2.5.2.3 Logical Data Model] to fit the Firestore NoSQL model we end up with the following structure:

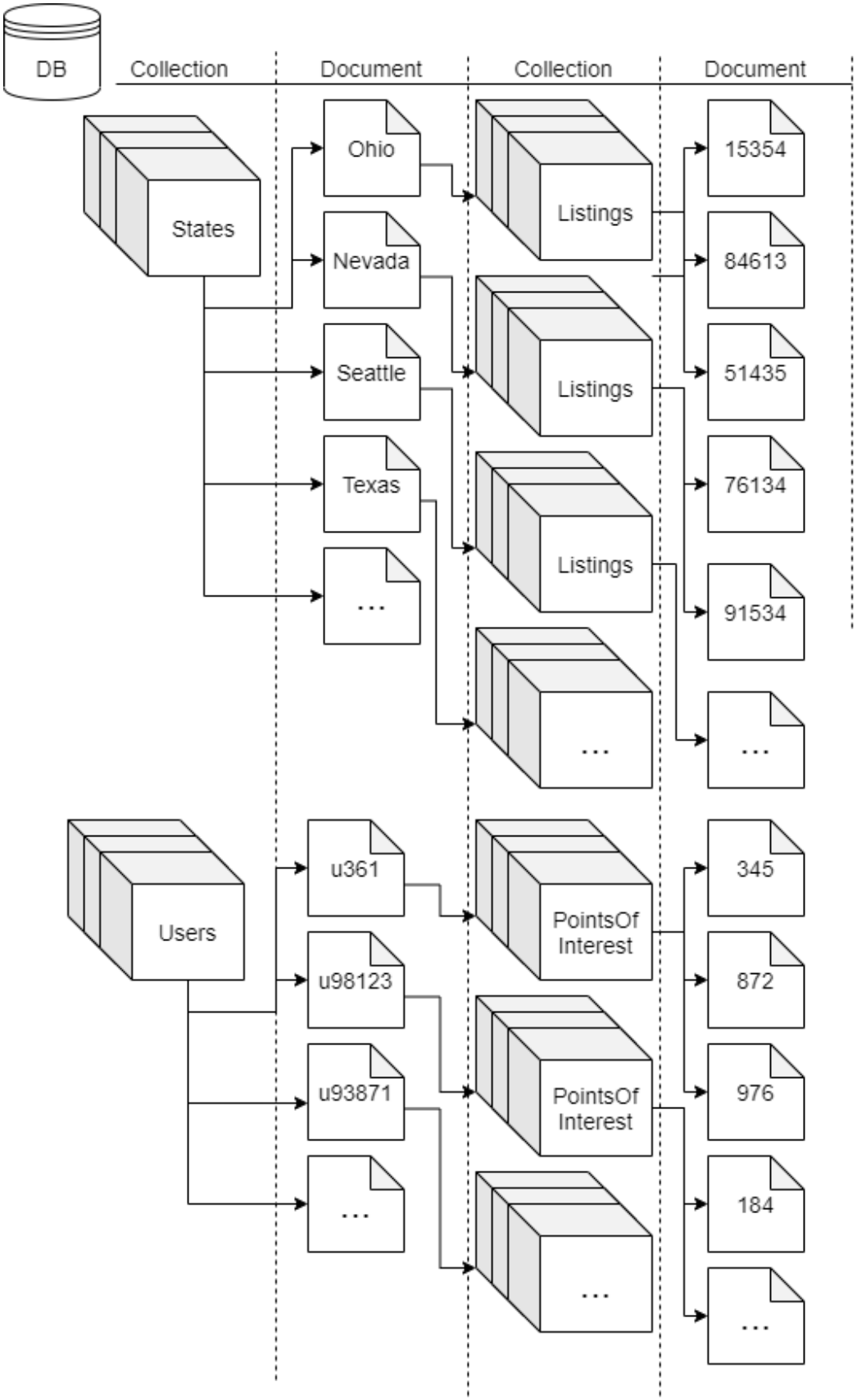


Figure 1: Application's database structure

Firestore gives us the ability to either query the database for a complete collection and receive all its documents or query for a particular document by its *ID*. We will utilize these functions later on.

3.2.3 Server

As mentioned before, the server side of our application is developed using the *Spring Boot* framework with which we manage to create an *API* exposing services ready to be consumed by clients. It is a complete autonomous application that doesn't rely on external systems to run and can be characterized as *RemoteProcedureCall* ***(RPC)***. Looking at the internals of our *Spring Boot* app we identify the following layer abstractions:

- **Controller/API Layer**: Responsible for creating the API endpoints and handling requests. *Spring* makes it so all classes in the Controller Layer are annotated with **@Controller**.
- **Service Layer**: Classes that contain the business logic of our application and are used as a means to keep coupling minimal. Similarly with the Controller layer, all classes in the Service Layer are annotated with **@Service**.
- **Model Layer**: Model classes implement data structures used by the application universe
- **Data Transfer Object or DTO Layer:** The layer containing classes representing objects that carry data between processes.

These layers' intercommunication can be understood with the following diagram:

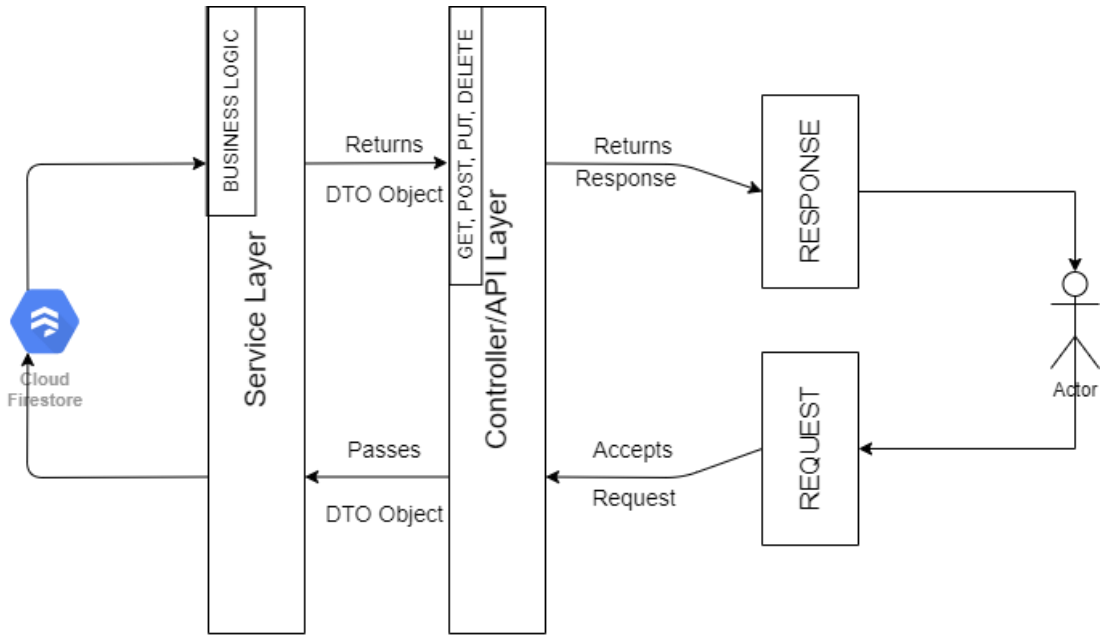


Figure 2: Spring Boot Application Layer Abstraction

3.2.3.1 Configuration

Since the server is the one that's making fetching and storing requests in the database, we need to establish a connection that's going to utilized by the rest of the app. *Spring Boot* works with what's called *Beans* where each Bean is instantiated on runtime. Basically, *Spring* allows an object to define its dependencies without actually creating them. It retrieves them from what's called an *Inversion of Control Container,* or *IoC Container* for short, that has been passed the proper configuration metadata. So, we create a configuration class annotated with @Configuration telling *Spring* to instantiate a connection with *Firestore* that will be used across all of the application.

3.2.3.2 Controller/API Layer

We move on to take a deeper look into the controller layer of the spring boot application. This layer's primary purpose is to expose our API's endpoints and handle incoming requests by parsing and forwarding them to the service layer. There are currently 4 controllers defined:

1. **StateController:** In charge of anything that has to do with the *States* Firestore collection.
2. **QueryMetadataController:** Responsible for the *QueryMetadata* Firestore collection.
3. **GeocodingController:** Receives requests of forward and reverse geocoding.
4. **UserController:** Receives requests from user generated actions.

These controllers expose the following endpoints:

API Endpoints (/api)			
State Controller ()	**QueryMetadata Controller ()**	**Geocoding Controller (/geocoding)**	**User Controller (/user)**
/states	/queryMetadata	/forward /reverse	/create/{uid} /{userUID} /{userUID}/poi/list /{userUID}/poi/createandadd /{userUID}/poi/add /{userUID}/poi/{poiUID}/delete /{userUID}/poi/batch/delete /{userUID}/poi/{poiUID}/edit /{userUID}/warehouseify

Table 5: API Endpoints

In more detail, these endpoints expose the following functionalities:

Endpoint Functionality			
Endpoint	**Controller**	**Method**	**Operation**
/api/states	StateController	GET	Returns the contents of the "States" collection in JSON format
/api/querymetadata	QueryMetadataController	GET	Returns the contents of the "QueryMetadata " collection in JSON format
/api/geocoding/forward	GeocodingController	POST	Performs forward geocoding on an address
/api/geocoding/reverse	GeocodingController	POST	Performs reverse geocoding on a pair of coordinates
/api/user/create/{uid}	UserController	POST	Registers a new user
/api/user/{userUID}	UserController	GET	Returns user's profile
/api/user/{userUID}/poi/list	UserController	GET	Returns a list representation of user's points of interest
/api/user/{userUID}/poi/createandadd	UserController	POST	Creates and adds a new point of interest to the user
/api/user/{userUID}/poi/add	UserController	POST	Adds a new point of interest to the user
/api/user/{userUID}/poi/{poiUID}/delete	UserController	GET	Deletes a point of interest from a user
/api/user/{userUID}/poi/batch/delete	UserController	POST	Deletes many points of interest from a user
/api/user/{userUID}/poi/{poiUID}/edit	UserController	POST	Edits the attributes of a point of interest
/api/user/{userUID}/warehouseify	UserController	POST	Searches for listings satisfying a user query

Table 6: Endpoint functionality

The class diagram of the controller layer relative to the other packages is:

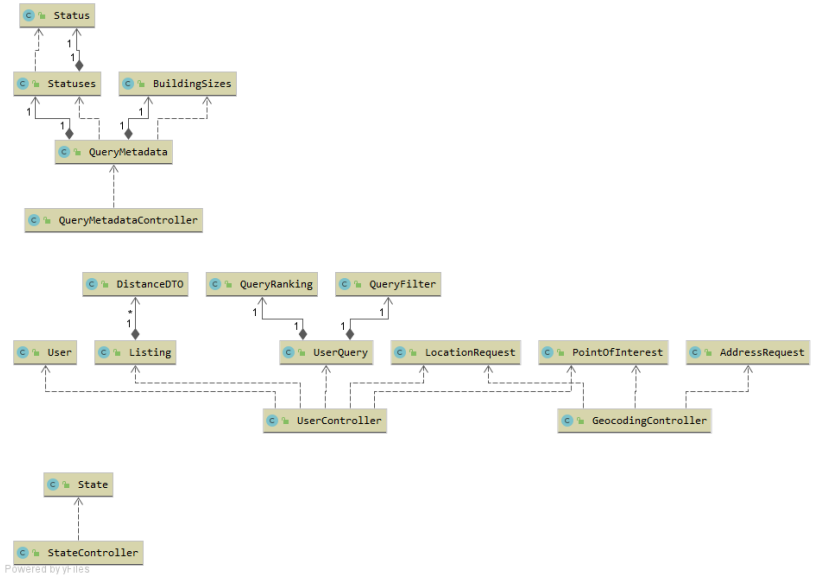


Image 6: Controller Layer Class Diagram

The composition of the *Controller* classes with the *Service* package is visible as well as the dependency unto the *DTO* package. The connection to the *DTO* package happens due to the *deserialization/serialization* that takes place inside the *Controller* classes. When a request arrives, *Spring* deserializes its body (if there is one) into a DTO object and passes it as a parameter to the correct controller method. Vice versa, serialization takes place during the building of the response by converting a DTO object to JSON. Note that although our application only handles JSON data, these operations are not exclusive to converting JSON back and forth.

3.2.3.3 Service Layer

The service layer does all the heavy lifting in the application by performing all the expensive computations and implementing all the business logic. For each service, its dependencies are shown in the diagrams below:

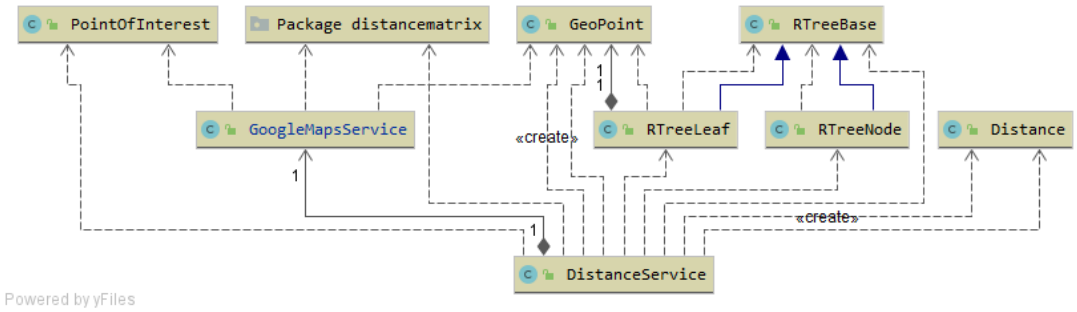


Image 7: DistanceService Dependencies

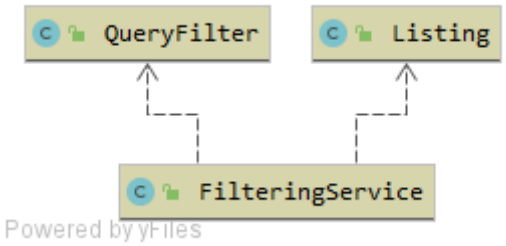


Image 8: FilteringService Dependencies

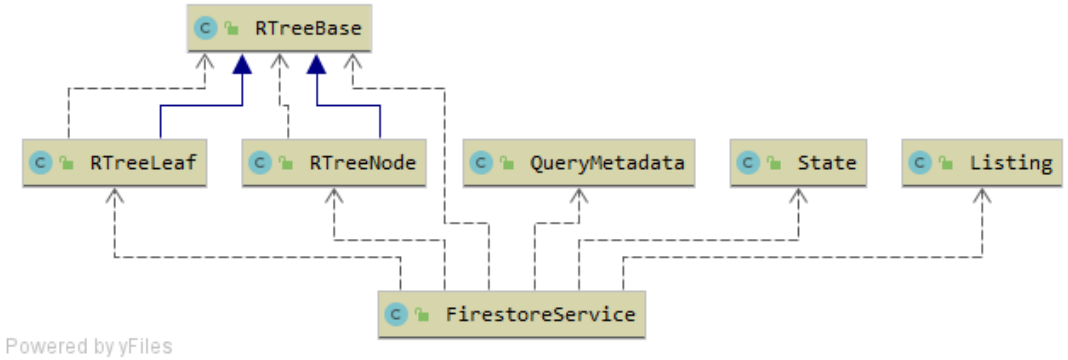


Image 9: FirestoreService Dependencies

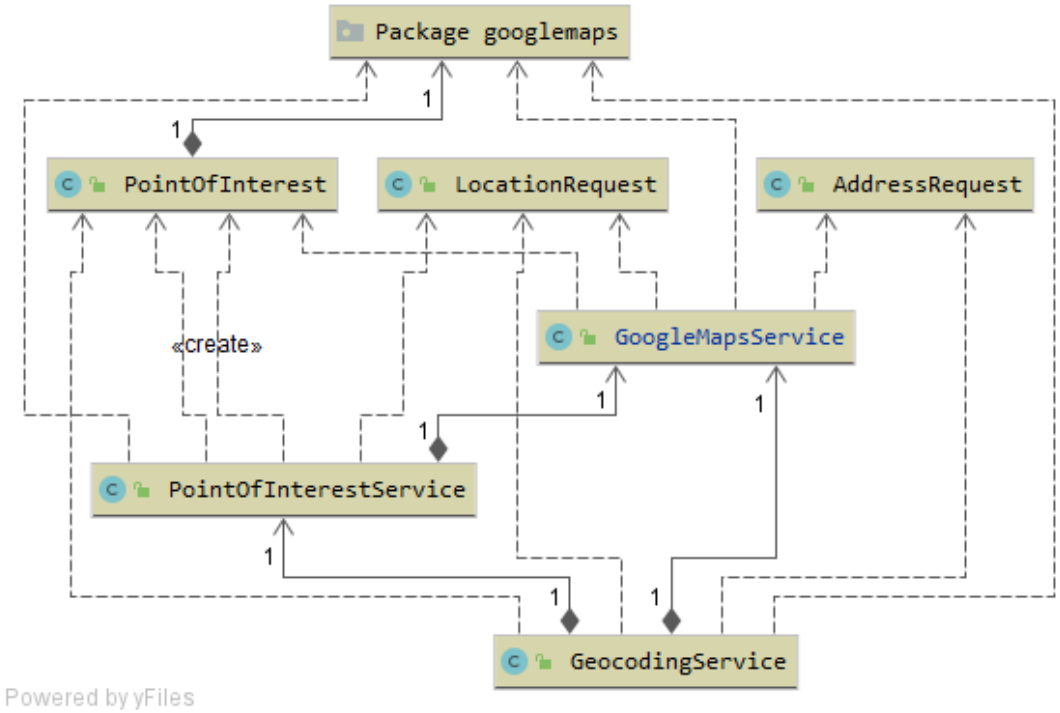


Image 10: GeocodingService Dependencies

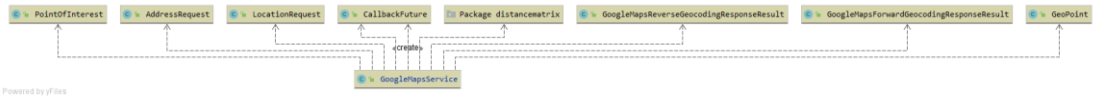


Image 11: GoogleMapsService Dependencies

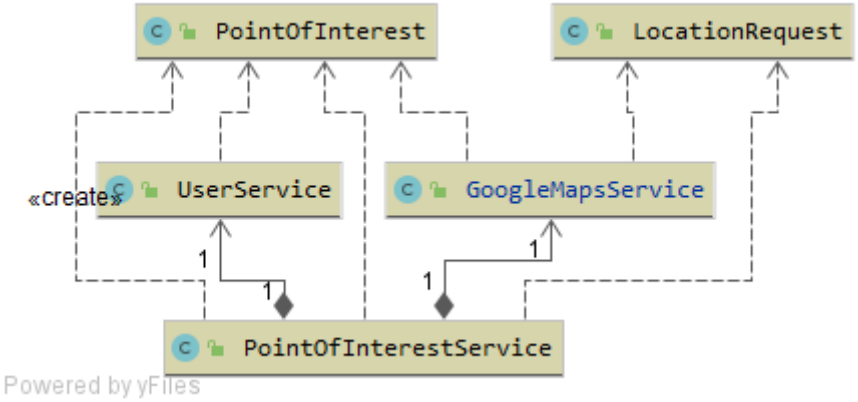


Image 12: PointOfInterestService Dependencies

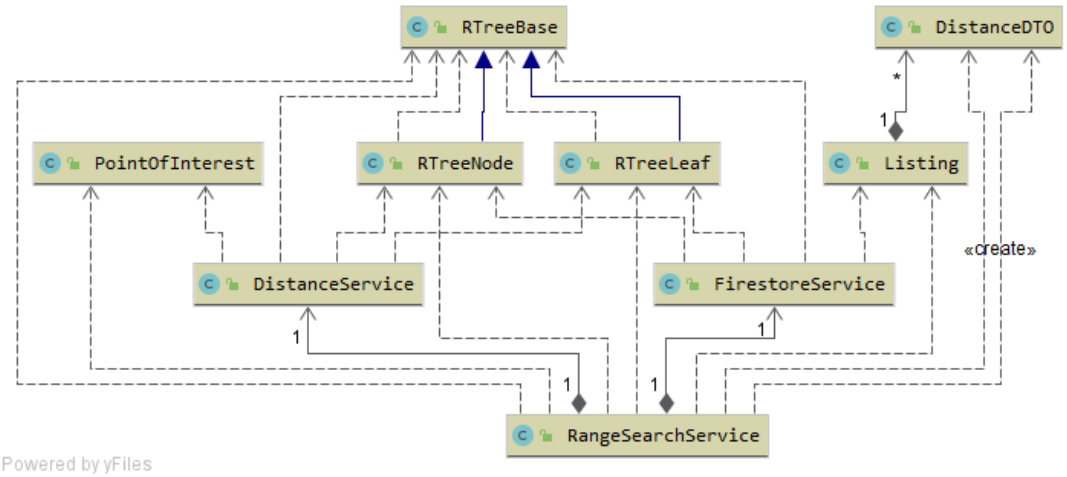


Image 13: RangeSearchService Dependencies

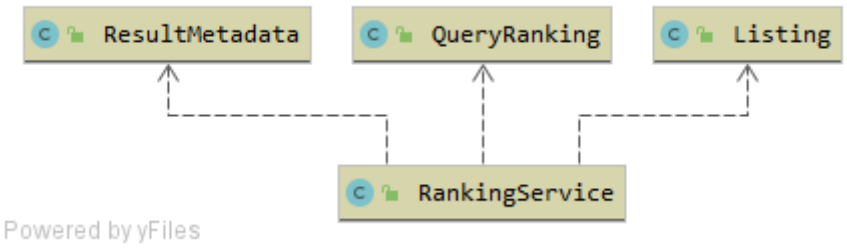


Image 14: RankingService Dependencies

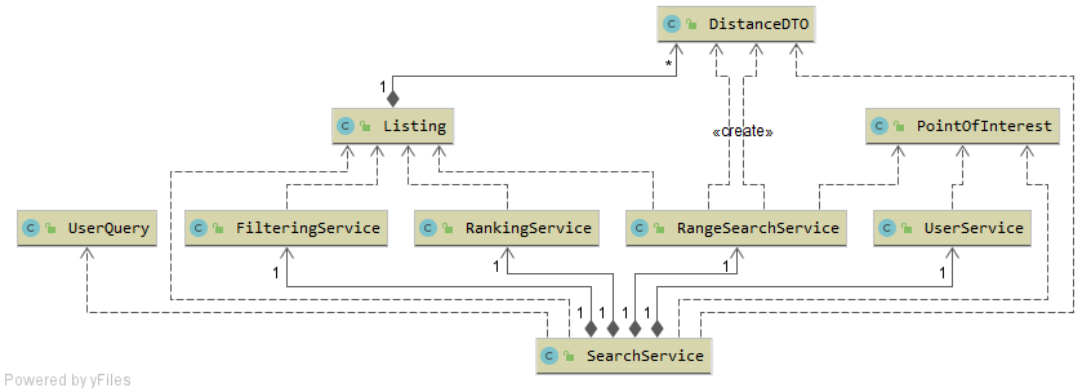


Image 15: SearchService Dependencies

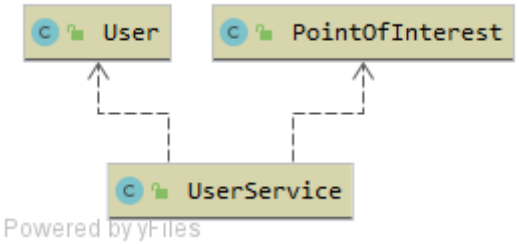


Image 16: UserService Dependencies

The service classes form compositions with each other and in particular with services that communicate with external APIs. We opted for designing different services based on their operational purpose with their name indicating the reason why they were created. We have the following services:

- **FirestoreService:** Contains methods accessing the *Firestore* database.
- **GoogleMapsService:** Contains methods communicating with the external Google Maps API.
- **DistanceService:** Contains methods for calculating the distance between two points.
- **RangeSearchService:** Performs range search on an RTree (more on that later)
- **FilteringService:** Filters the results obtained from the range query.
- **RankingService:** Scores the given results based on their attribute values.
- **GeocodingService:** Performs the forward and reverse geocoding.
- **UserService:** Contains user-related methods.
- **PointOfInterestService:** Contains methods related to Points of Interest manipulation and creation.
- **SearchService:** Head of organizing the search for listings given a query.
- **CallbackFuture:** Utility class/service used to make asynchronous API calls.

Each class and method maintain the single-responsibility principle.

3.2.3.4 Model Layer

Our model layer, while poor in the number of classes, contains two which are used inside the domain of the *Spring Boot* application.

- **ApiError:** Used internally for keeping info of an occurred error.
- **Distance:** Used during the calculation of the distance between two points.
- **ResultMetadata:** Holds summarization data on the finalized result set. Used by the ranker.

3.2.3.5 Error handling

Within our API we have added the utility of custom error reporting primarily for processes that require communication with external systems such as the Firestore database or any of the Google's APIs. Specifically, the API returns the following codes:

API Error Codes		
Error Condition	**HTTP Response Code**	**Response Body**
Resource was not found in the database	404	${resourceName} resource with id {${uid}} was not found.
Service is unavailable	503	Server encountered an unavailable service.
Invalid geocoding arguments	400	Wrong or missing geocoding arguments.

Table 7: API Error Codes

The two first errors are of catch-all type, meaning that their structure, namely error messages, are created dynamically able to be thrown by any resource or service respectively. So, when a user or a point of interest is not found, the same type of error is thrown but with a different debug message to indicate what was the exact issue. Similarly for the unavailable services. This allows for new resources or services to be added with ease.

3.2.3.6 Caching

Spring allows us to cache method calls by storing a key-value pair the parameters passed to a method and the return value of that method. By default, *Spring* creates a ConcurrentHashMap to store these pairs and that's what our application is using. This technique, while it could be proven dangerous if done without proper care, results in a great optimization of our processing time as we can often skip doing complex calculations or accessing the database and paying a hefty fee in waiting time. A point should be made that we MUST not cache data that are subject to frequent change, such as user data, as that would lead to data stagnation.

3.2.4 Client

The client, built using the React JS framework, is designed to provide a fast, responsive and user-friendly experience. React builds one-page apps and dynamically renders new content on that one page allowing for faster content loading. In this section, we will try to analyze in summarization its inner workings and design, starting by present its general layout:

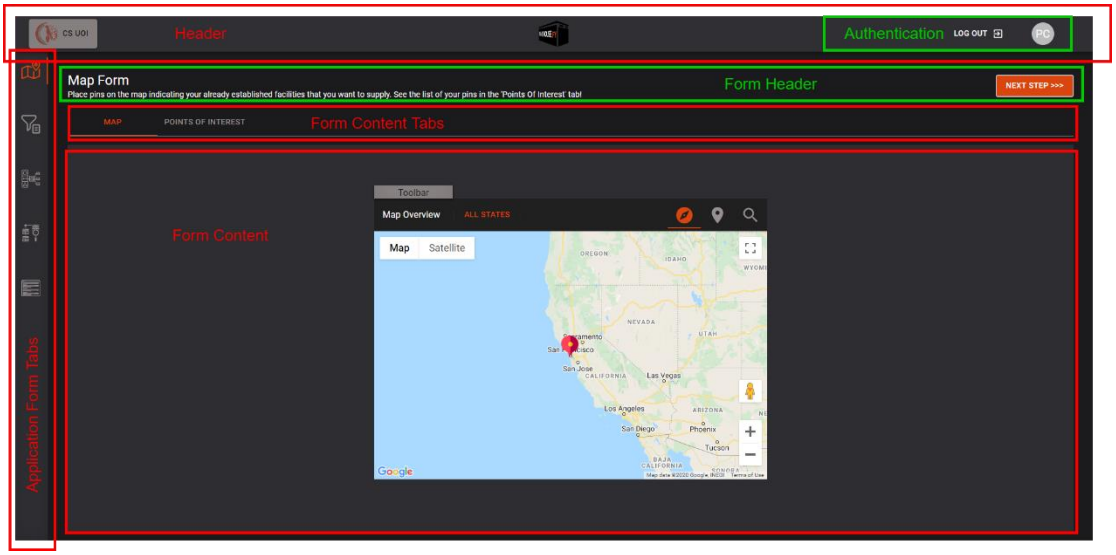


Image 17: Client Layout

This is the core layout that's followed around the application heavily utilizing tab navigation.

3.2.4.1 Authentication

User authentication is taking place in the client side of our app by using the *Firebase Auth* service with the provided sign-in method of email & password. When a user creates a profile the system directly contacts the external service with no messages going through the server. While routing authentication through the backend server was possible, we chose to avoid that in favor of a more simplistic approach.

Authentication grants us the ability to prevent viewing content that's not intended for some users as well as creating user profiles for storing user data. The part of not viewing a particular resource when not authorized is called "route guarding" and is implemented in our application. React apps, since them being single-paged, use route pathing where each path corresponds to displaying different content. Usually, these paths separate different operations as for example in our app, we have:

- "/login" and "/signup" route path for rendering the login and signup page respectively,
- "/" route path, which is the root of our *React* application and serves as a landing page,
- "/main" route path for rendering the main application forms.

Now, we want to be able to only show the "/main" content to authorized users and that's where route guarding comes in by checking if the user trying to access said content is logged in or not and redirecting them to the login page in the latter case. Vice versa, when

a user is logged in, we don't want to show them the login and signup pages, as that would create all sorts of mess, so we mirror the principle stated above by not allowing authorized users in routes that are intended for non-authorized ones. These two concepts are implemented in the **PrivateRoute** and **PublicRoute** components of our application respectively. These two components encase whatever content we want to "protect" from being accessed by the wrong subset of users.

3.2.4.2 Data persistence

The UI makes use of many external data that are stored inside the application's database and, although, they are considered dynamic in the sense that they are not hard-coded inside the client code, they do not get updated much often. In fact, the only time they might get updated is when a big patch for application comes along. So, we could argue that it is a kind of static data loaded dynamically, since the client makes API calls to the backend to fetch them from the database on initial page load. This data is mostly used as UI data in dropdown menus and are not in any way sensitive. So, we come to the realization that we could store them in local storage on a per session basis so that the user can exit and return the application and not experience the somewhat longer loading time that comes with the initial visit.

You may ask "But what consists a session?" and to that the answer is that we made the decision, considering the scope of the project, to be the span a user remains logged in to the system. *Firebase Auth* gives us the ability to remain authenticated even after the browser window closes (this is subject to change and would be a nice addition for the user to be able to select "Keep me logged in" or not).

3.2.4.3 Latency Compensation

User actions affecting user-related data, such as editing or deleting a point of interest, require the client to make the appropriate call to the backend API in order to actually have an effect, since the entry in the database needs to be modified. Such actions take time to be completed due to latency and hinder the user experience when the resulting effect takes time to be communicated to the user through an updated UI state.

In order to combat that, we have implemented a way for changes to take effect on a local scale first and then be validated when the server matches these changes. When a user tries to edit a point of interest the changed state gets immediately rendered as if it happened immediately. At the same time an API call is dispatched asking for the same thing to be done on the database entry. The server accepts and processes the request returning if it was carried out successfully or not. If it was, nothing changes, the user has

been notified that all was well and the system moves on. In case the request was unsuccessful, the client reverts the changes back to the pre-edit state and notifies the user that a rollback was completed.

One might think that this technique could lead to a performance hit, since we keep a copy of the previous state, but being careful to only store the part that was edited instead of the whole data structure results in minimal use of local memory storage use. For example, when editing a point of interest, we keep only a copy of the to-be-edited PoI instead of the whole list of PoIs before the changes take effect. If the changes get rejected, the old state is appended on the PoI list overriding the edited state and the memory space is freed.

3.3 Business Logic

The primary goal with the business logic of our application, as described in [3.1.2 Business Logic], was to create a functionality that would be both interesting and useful. This was achieved by manipulating our dataset that consisted of real-life listings, as well as creating an efficient and fast way to extract the wanted information.

3.3.1 Dataset

The dataset that would serve as the backbone in our database was gathered by scraping listings from the web using a web-crawler, were 1 listing equaled 1 commercial property. The data that was initially amassed numbered to 25.000 listings but after careful parsing and removal of unusable data we ended with ~19.000 all spread across the contiguous USA.

We proceeded to shape the dataset by removing listings missing critical data such as location (latitude, longitude), price and title, replacing missing values with a default value, converting field names to camelCase attribute names and field values with string-ified numbers to actual numbers. Each listing is referenceable by its actual id.

Plotting all the datapoints we get the following:

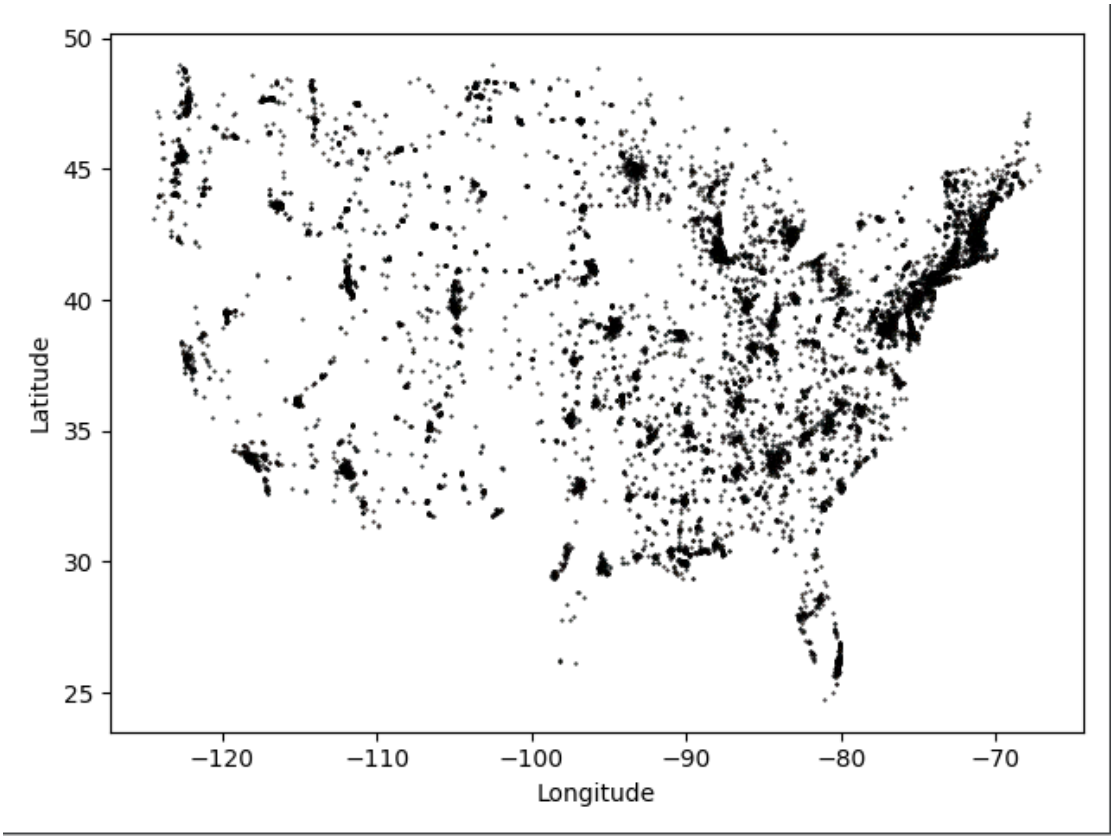


Image 18: Dataset plot

3.3.2 Algorithm

The process of finding all the listings that fall inside an overlapping area is solved by performing a **range query** on our dataset. The logic behind a range query isn't complicated, listings that fall in the designated area have coordinates that are inside the radiuses of the user's points of interest. Of course, checking every listing against every point of interest is not efficient and so, we end up needing a data structure that will help us achieve low searching times.

That structure is an **R-Tree** and we create it by bulk-loading it our dataset. We use the **Sort-Tile Recursive (STR)** algorithm to create the R-Tree from our data. With a leaf size **B = 10** we get the following structure:

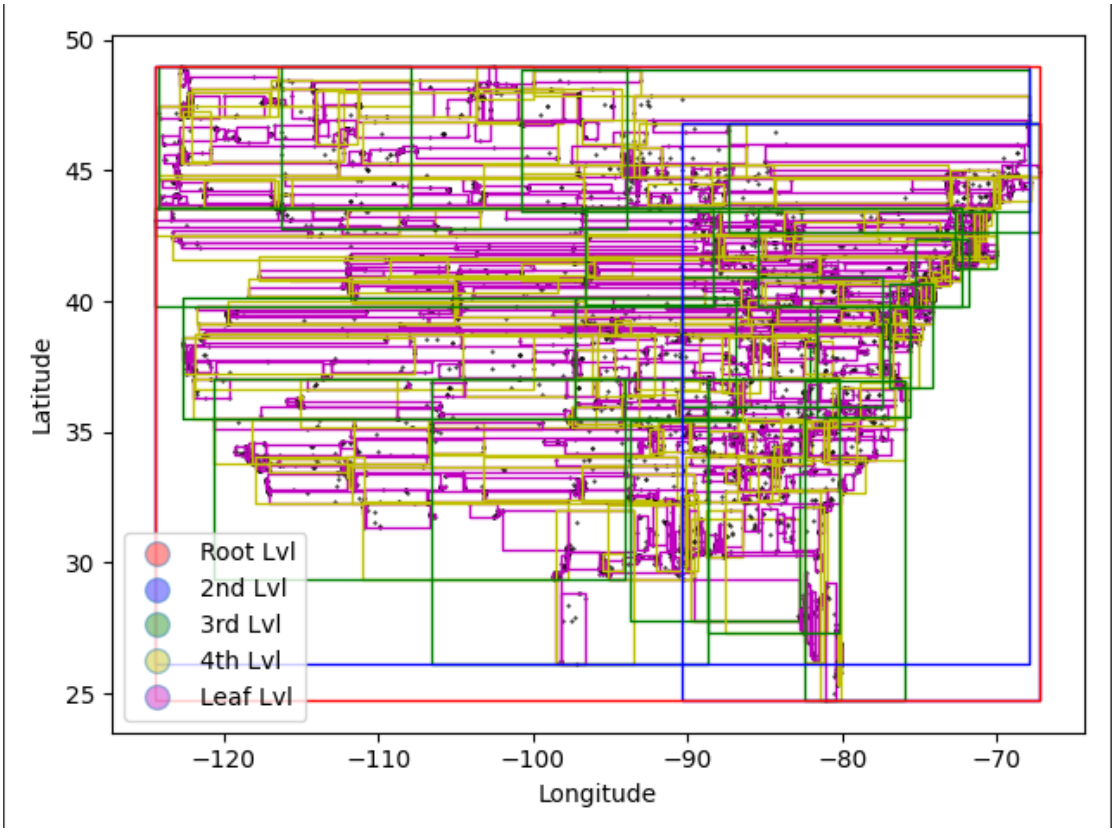


Image 19: R-Tree structure of dataset for B=10

The STR bulk-loading method was implemented in python 3.8 and the resulting structure stored in a dictionary with each node and sub-node represented by a sub dictionary till we reach the leaf nodes which point to entries containing the ID and location of each listing within them. This is compatible to be uploaded to Firestore, as it is of JSON format, with each node represented by a *document* and its children nodes contained in a *sub-collection*. The node documents, since they represent **Maximum Bounding Boxes (MBRs)**, are described by two attributes, one pointing to the location of their minimum and the other to the maximum dimensional values, or translating it to geographical data; their southwestern and northeaster points.

The range query algorithm was implemented inside the RangeSearchService class of the *Spring Boot* application and it follows the following steps:

Algorithm 1: Range Search		
Input	:	PointsOfInterest userPoi, RTree root
Output	:	List of listings
Step 1	:	results ← Initialize empty list
Step 2	:	currentNodes ← root
Step 3	:	While currentNodes is not empty:
Step 4	:	For each poi in userPoi:
Step 5	:	If (currentNodes == RTreeNodes):
Step 6	:	distances ← haversineDistances (poi, currentNodes)
Step 7	:	Else If (currentNodes == RTreeLeafs):
Step 8	:	distances ← drivingDistances (poi, currentNodes)
Step 9	:	currentNodes ← filterOutNodes (poi, currentNodes, distances)
Step 10	:	If currentNodes is not empty:
Step 11	:	If (currentNodes == RTreeNodes):
Step 12	:	currentNodes ← unpackNodes (currentNodes)
Step 13	:	Else If (currentNodes == RTreeLeafs):
Step 14	:	results ← unpackLeafs (currentNodes)
Step 15	:	break;
Step 16	:	return results

Algorithm 1: Range Search

The algorithm is able to find all those listings that satisfy the constraints with one downward traversal of the R-Tree checking at each level if the candidate MBRs satisfy the user query.

3.3.2.1 Distance calculation

The way a candidate node is checked against all the user points of interest is by executing the *MINDIST* algorithm between the node's MBR and each poi iteratively. The distance between an MBR and a point of interest is calculated using the ***Haversine* formula** until a leaf node is reached where using the ***Google Maps Distance Matrix API***, we find the **drivable** distance between the location of the actual listing and the point of interest.

To illustrate the difference between the two types of distance we present the following:

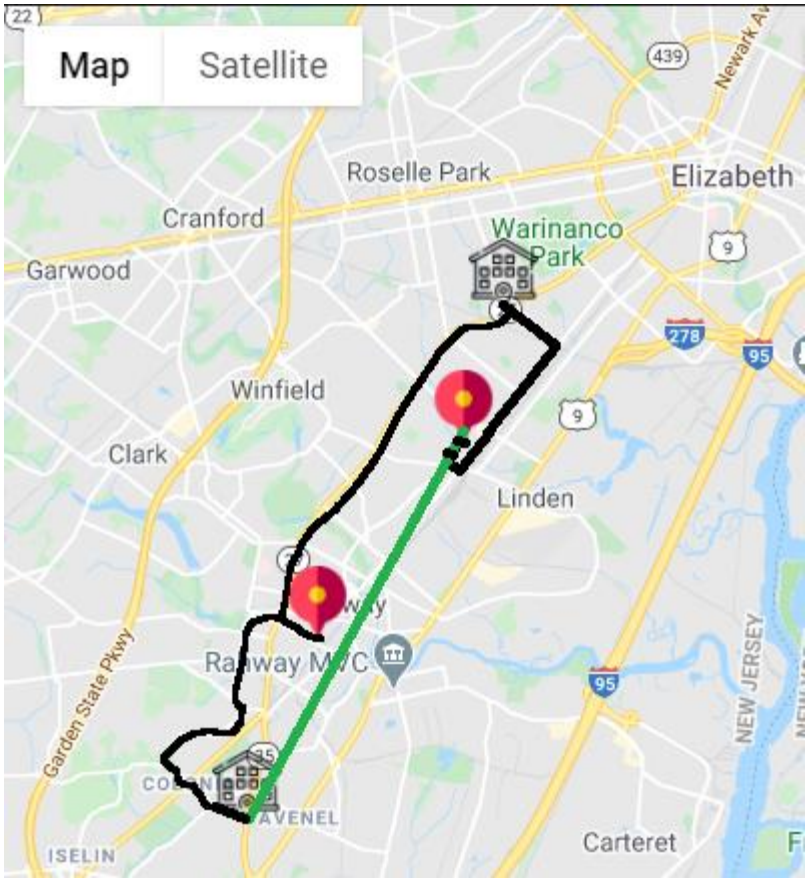


Image 20: Distance calculation illustration

Assume that the **black** lines represent drivable, calculable routes that have been evaluated using the *Distance Matrix API* and the **green** line represents a connection between a point of interest (red balloon marker) and a southwestern listing (warehouse icon marker) that has not been evaluated by the external API due to reasons beyond our ability to influence (Google lists the reasons on their documentation pages). Instead of dismissing the listing immediately, we calculate the distance using the *Haversine* formula as a fallback mechanism. We mark the calculation as "ESTIMATE" to let the user know that it may not have been an accurate result but to the best of our knowledge, it is. On the opposite side, listings that we are absolutely sure about, we mark them as "EXACT".

3.3.2.2 User customization

After executing the range query and retrieving the resulting listings, user customization takes over by filtering these results and then ranking the remaining ones. The whole process of searching, filtering and ranking takes place in the **SearchService** class of the *Spring Boot* application.

3.4 Testing

All the unit and integration tests are done in self-isolation while the system-wide tests were conducted on a local production build of the application in order to secure system isolation and integration.

We identify the following steps:

1. **Unit testing:** Unit testing revolves around the *Service* layer and testing its components in isolation
2. **Integration testing:** Combine parts that make up a bigger part and test it in self-isolation.
3. **System testing & Requirements assessment:** During this phase we test the system as a whole and create a traceability matrix tracking the number and type of tests conducted relating to one specific requirement mentioned in the SRS.

Since our application has an API, we want to be able to test it to see its effectiveness in operation. To achieve this, we test for the following:

- If the API listens on all endpoints for HTTP requests
- If the API is able to deserialize all input
- If the API is able to call the business logic (*Service* layer)
- If the API is able to serialize the output

We create integration tests on all Controllers of the API, mocking their dependencies when needed in order to test in isolation. For example, we can see if the business logic is called by the Controller just by mocking the *Service* class and looking if it has been called with the right arguments.

Moving on, we have prepared two matrices for system-wide testing that refer to the **Test Cases** we used as well as the **Traceability Matrix** of those test cases relative to the requirements laid out in the SRS. Of those test cases, the system passes on the majority of them with the failing test cases having a reason annotated next to them for further processing.

3.5 Deployment & Use

The application is packaged as a whole in a single *.jar* file and is deployed on an *EC2* instance on the Amazon Web Servers using Elastic Beanstalk. The application can be seen here. Since it is a web application, there is no need for any initialization values.

We will create a walkthrough on how to utilize the application:

1. Navigate to the sign-up page by clicking the “Sign Up” button located on the top-right corner of your screen.
2. Create an account by filling in the fields and submitting. You will be redirected to the landing page.
3. Enter the “/main” subdirectory of the application by clicking the “Start” button located on the right side on the landing page.
4. Navigate to the Map Form and pan to a desired location. (Tip: New-York is relatively guaranteed to contain multiple listings)
5. Select the second tab on the map toolbar (the marker icon)
6. Left-click on the map to place point of interest.
7. Head-over to the Points Of Interest tab, inside the Map Form, and edit the added points of interest to your preferences.
8. Navigate on the Query Filtering Form by clicking the second tab on the left vertical sidebar.
9. Adjust the filters according to your preferences and hit the “Store Preferences” button.
10. Navigate on the Query Ranking Form by clicking the third tab on the left vertical sidebar.
11. Adjust the ranking biases according to your preferences and hit the “Store Preferences” button.
12. Navigate on the Overview tab by clicking the fourth tab on the left vertical sidebar.
13. Have a final look on all your input values and hit the “Search Now” button located on the right side of the panel.
14. View the results on the Results tab. The results are split into two categories “For Sale” and “For Lease”. Click on the right-sidebar on the Results tab with the “MAP” label and view the results on the map relative to your points of interest.

3.6 Maintenance and Scalability

The application is generally built with modularity in mind meaning the ability to add new services should be straightforward and not affect other parts of the code. On a future update we should look into refactoring and decoupling the dependence of the

Service layer from the *DTO* layer by introducing equivalent *Model* classes. Currently, any change on a *DTO* object will result in a change in all of the *Service* classes that use it which is DEFINETELY an unwanted behavior.

In terms of data storage and persistence, the application is set up to support unlimited users and user data as long as *Firestore* continues to do so. There is no need for maintenance in order to support scalability.

Furthermore, functionalities like query filtering and ranking should be made more modular because currently they do not allow for more filtering or ranking fields.

Chapter 4. Experiments

In this section, we will present results based on testing done on the application to see how it performs under load as well as the performance of our business logic.

4.1 Experimentation Purpose

We aim to test scalability by measuring the performance of our application under load and to have sufficiently good response times. These scalability tests refer to scaling based on concurrent user requests, where multiple clients are sending our server API different types of requests, scaling our business logic based on the number of input values.

These tests will be performed on the following environment; on a local production build of the application where latency between server and client is zero (localhost) and the host machine is running on a 4.4GHz processor.

The tool we will be using is going to be the open-source application *Apache JMeter*, designed to measure performance and to load-test functional behaviour. Note that we won't be comparing endpoint performance which carry out operations such as WRITE, DELETE and UPDATE on the user profile, as that would require multiple different user accounts and wouldn't provide us with any more useful information than what testing other endpoints will.

4.2 Experimentation Results

In all of our experiments we test against the same 3 endpoints:

- **/api/states**: Returns a large collection retrieved from Firestore db
    - Request Headers:
      Connection: keep-alive
      Host: localhost:5000

User-Agent: Apache-HttpClient/4.5.12 (Java/13.0.2)

- Request Body:

  GET http://localhost:5000/api/states

  GET data:

  [no cookies]

- **/api/querymetadata**: Returns a small collection retrieved from Firestore db
  - Request Headers:

    Connection: keep-alive

    Host: localhost:5000

    User-Agent: Apache-HttpClient/4.5.12 (Java/13.0.2)
  - Request Body:

    GET http://localhost:5000/api/querymetadata

    GET data:

    [no cookies]
- **/api/user/{userUID}/warehouseify**: Returns the results found by the main searching business logic. As a baseline, we will be passing two points of interests and will not be customizing filtering or ranking on the request body.
  - Request Headers:

    Connection: keep-alive

    :

    Content-Type: application/json

    Content-Length: 344

    Host: localhost:5000

    User-Agent: Apache-HttpClient/4.5.12 (Java/13.0.2)
  - Request Body:

    POST

    http://localhost:5000/api/user/hJOSMuP67FMk0lunMa56Ga9rx5l2/warehouseify

    POST data:

    {"queryFilter":{"buildingClass":"","priceRangeFrom":0,"priceRangeTo":9007199254740991,"propertyType":"","state":"","status":"","buildingClassCheckbox":false,"propertyTypeCheckbox":false},"queryRanking":{"buildingClassBias":{"all":"1"},"buildingSizeBias":{"all":"1"},"distanceBias":{"all":"1"},"lotSizeBias":{"all":"1"},"priceBias":{"all":"1"}}}

    [no cookies]

- **/api/geocoding/reverse**: Returns the reverse geocode of a (latitude, longitude) pair. We will be passing such a pair on the request body.
    - Request Headers:

      Connection: keep-alive

      :

      Content-Type: application/json

      Content-Length: 64

      Host: localhost:5000

      User-Agent: Apache-HttpClient/4.5.12 (Java/13.0.2)
    - Request Body:

      POST http://localhost:5000/api/geocoding/reverse

      POST data:

      {

        "latitude": 36.2746858,

        "longitude": -119.7845888

      }

      [no cookies]

The last two endpoints pass through to parameterized functions and therefore require we send a response body along the POST request. On the searching endpoint we set as default two points of interest and no filtering or ranking and on the reverse geocoding endpoint we attach a pair of latitude-longitude coordinates.

1. We begin by setting a general baseline for each endpoint by 10 concurrent requests on each endpoint. These sets of requests are sent consecutively and NOT in parallel.

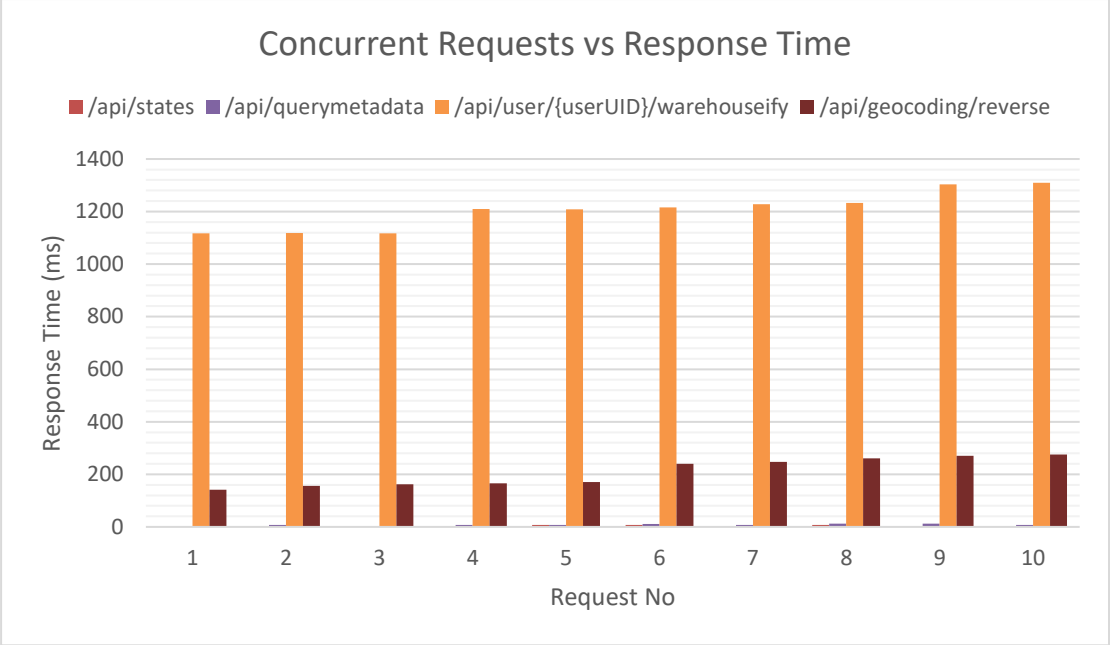


Figure 3: Sending 10 concurrent request to each endpoint consecutively

It is only logical that the more heavy the processing is, the higher will the response time be. The collection fetching that's being done through two of the endpoints is blazingly fast as their return value gets cached. It is particularily beneficial that these resources get cached, as they are unchanged throughout the application lifespan.

2. We send 10 concurrent requests to fetch a small database (querymetadata) collection to see how much does caching a resource decrease the response time of the server.

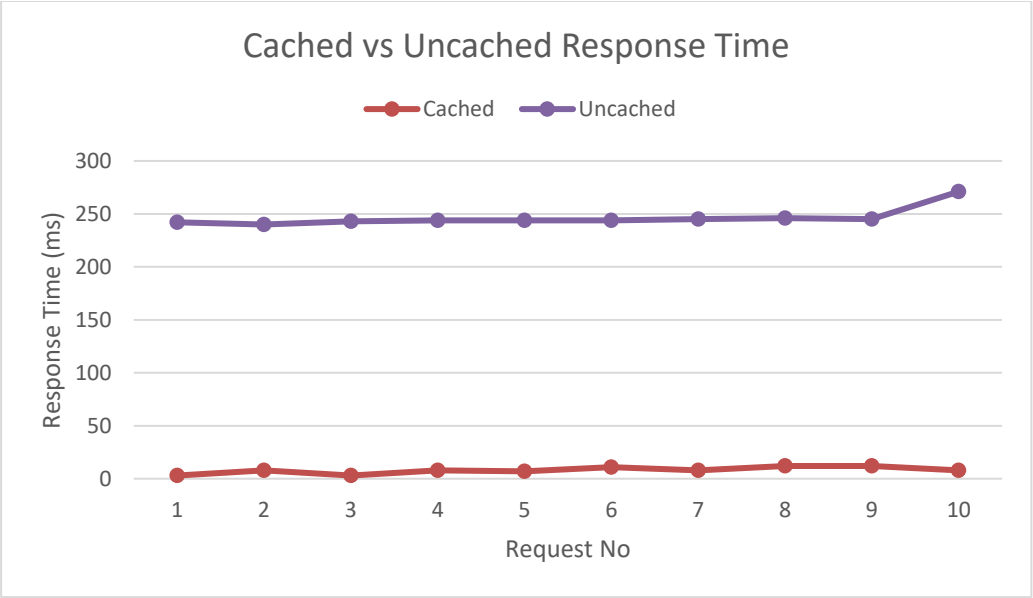


Figure 4: Sending 10 concurrent requests with caching or and off

Caching is saving us around 230ms of response time which is a really nice technique to save time during loading. (As a side note, the client when it initially loads requests some resources that are required to be rendered in the UI. While that happens, the system does not render anything in order to avoid user confusion. The point is that by caching and decreasing the response time, we, in turn, decrease initial loading times.)

The first test assumed that all concurrent requests would be directed at the same endpoint each time, which is not really the case in a real-world environment where requests would be of mixed type. Furthermore, *Spring Boot* uses *Tomcat* to deploy the server application and the default maximum thread pool size allowed by *Tomcat* is 200. Following the technique of Robust Boundary Value testing, we will test for the following concurrent thread values: [8, 40, 100, 168, 200, 232] with conducting 10 tests for each value and averaging the results. We are looking to compare the average response times.

3. 8 concurrent threads - 10 tests:

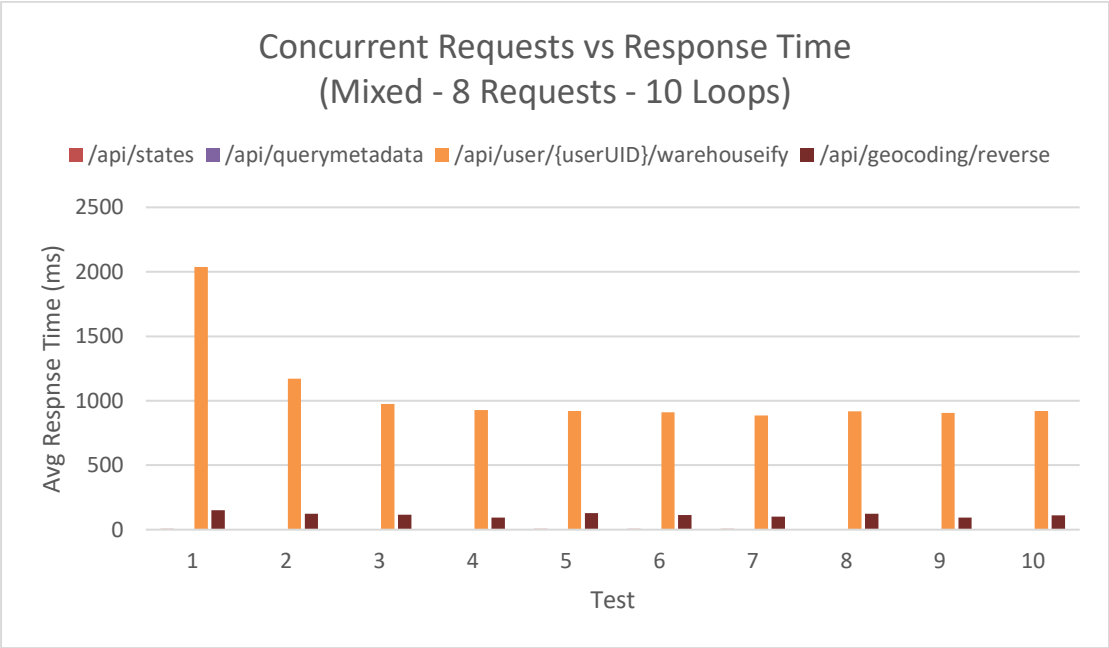


Figure 5: Sending 8 concurrent requests

Summary

Label	# Samples	Average Response Time (ms)
/api/states	20	5
/api/querymetadata	20	0
/api/user/{userUID}/warehouseify	20	1056
/api/geocoding/reverse	20	115

Table 8: Average latency of 10 tests sending 8 concurrent requests

We notice a declining trend, that will also become apparent in later tests, which can be translated as if the system "warms-up" after a period of idling. After "warming-up" the server is capable of handling traffic with ease.

4. 40 concurrent threads – 10 tests

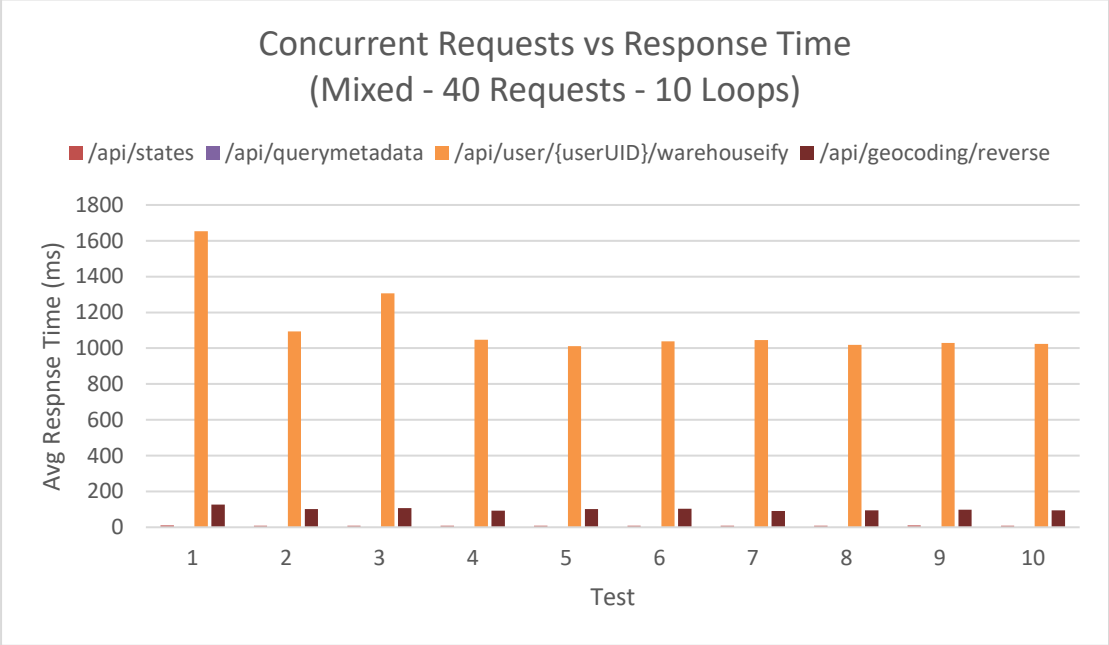


Figure 6: Sending 40 concurrent requests

Summary

Label	# Samples	Average Response Time (ms)
/api/states	100	9
/api/querymetadata	100	1
/api/user/{userUID}/warehouseify	100	1127
/api/geocoding/reverse	100	101

Table 9: Average latency of 10 tests sending 40 concurrent requests

5. 100 concurrent threads – 10 tests:

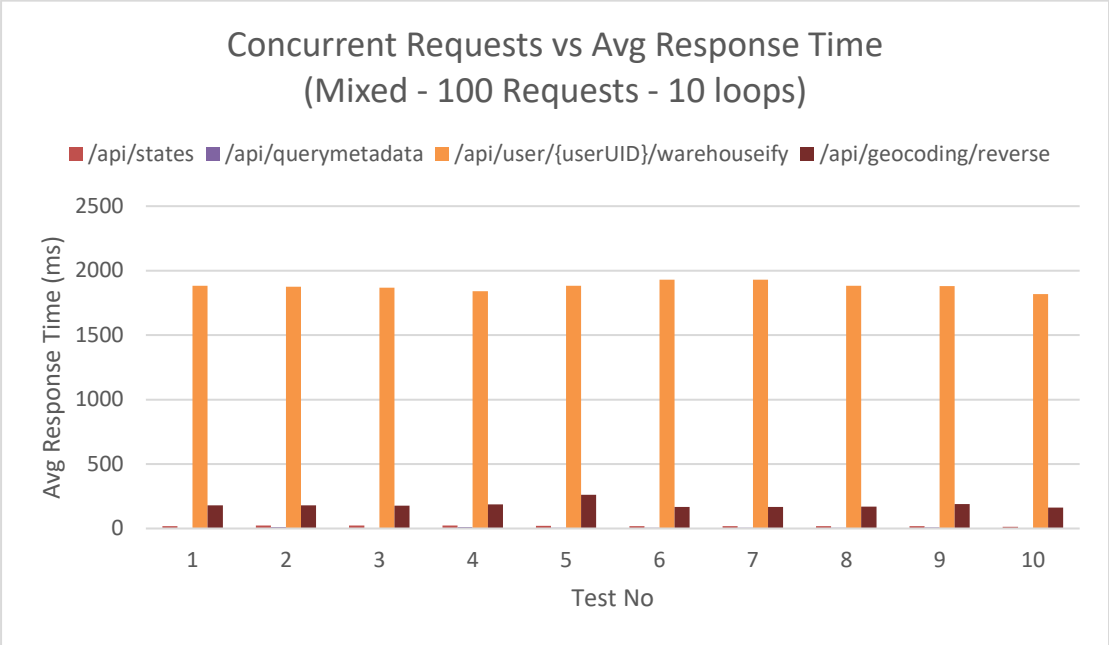


Figure 7: 10 tests sending 100 mixed concurrent requests each

The above tests give as the following averages response times:

Summary		
Label	**# Samples**	**Average Latency of Tests (ms)**
/api/states	250	19.6
/api/querymetadata	250	6.1
/api/user/{userUID}/warehouseify	250	1878.9
/api/geocoding/reverse	250	184.1

Table 10: Average latency of 10 tests sending 100 concurrent requests

Even when increasing the number of concurrent users to 100, the server can handle requests fine relatively to previous benchmarks. We move unto the next testing value:

6. 168 concurrent threads – 10 tests

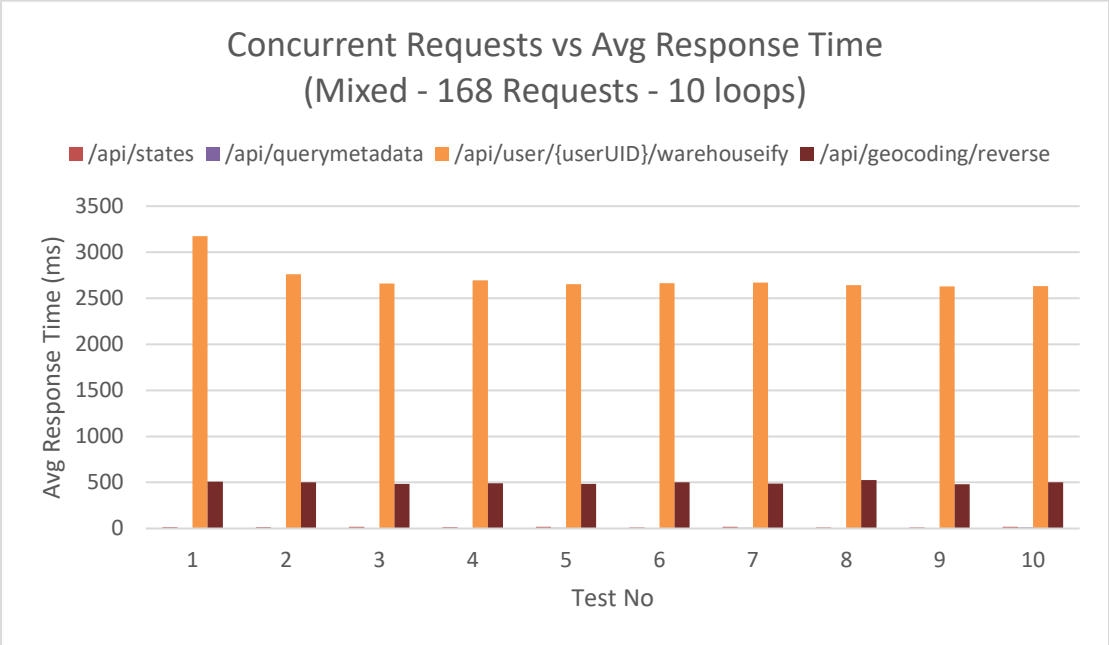


Figure 8: 10 tests sending 168 mixed concurrent requests each

Summary		
Label	**# Samples**	**Average Latency of Tests (ms)**
/api/states	420	15
/api/querymetadata	420	6
/api/user/{userUID}/warehouseify	420	2718
/api/geocoding/reverse	420	496

Table 11: Average latency of 10 tests sending 168 concurrent requests

The declining trend keeps going, albeit not as steep and that's due to the system reaching its limits slowly but surely. We go on the penultimate value and matching the max thread pool set by *Spring* and *Tomcat.*

7. 200 concurrent threads – 10 tests

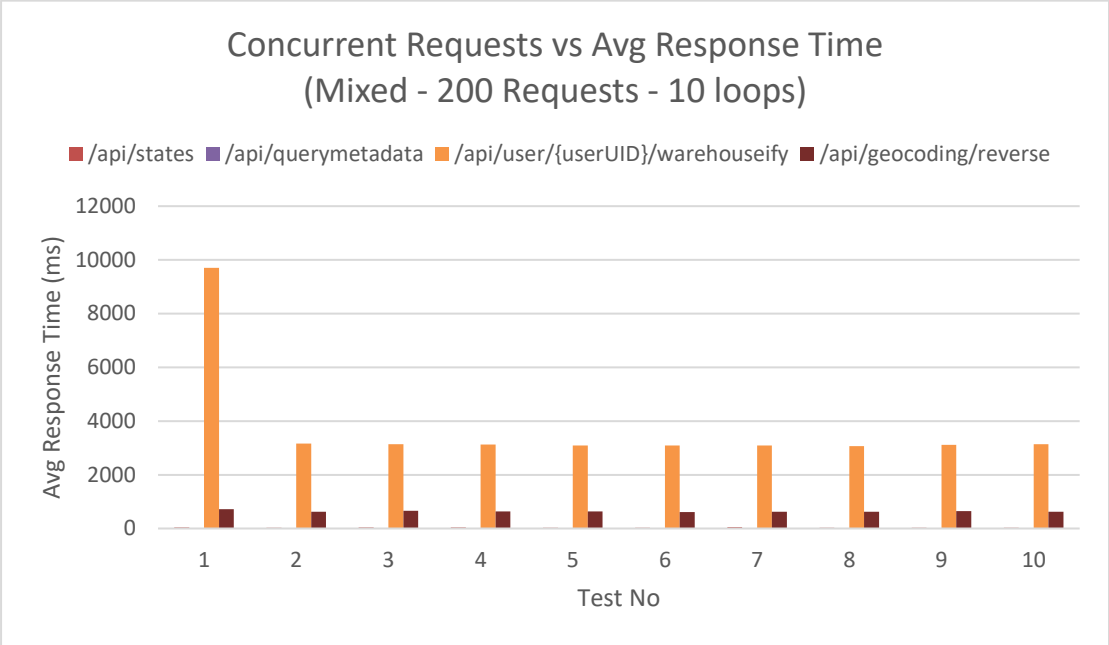


Figure 9: 10 tests sending 200 mixed concurrent requests each

These tests average the following:

Summary		
Label	**# Samples**	**Average Latency of Tests (ms)**
/api/states	500	33
/api/querymetadata	500	11
/api/user/{userUID}/warehouseify	500	3773
/api/geocoding/reverse	500	640

Table 12: Average latency of 10 tests sending 200 concurrent requests

Noticeable spike in average response time of the first test for the searching endpoint which is the highest we've seen till now. Beyond that, the application seems to be averaging significantly higher numbers than before, pointing that we are nearing the limit on concurrency. Finally, we are going ahead and exceeding the max-thread pool.

8. 232 concurrent threads – 10 tests

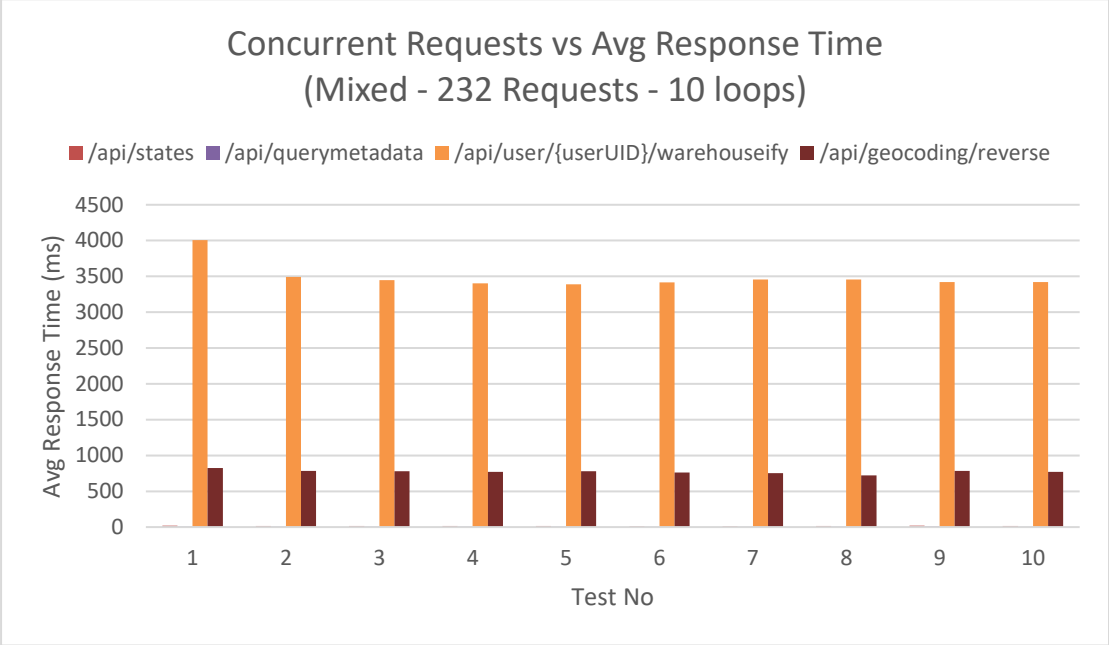


Figure 10: 10 tests sending 232 mixed concurrent requests each

Summary		
Label	**# Samples**	**Average Latency of Tests (ms)**
/api/states	580	16
/api/querymetadata	580	6
/api/user/{userUID}/warehouseify	580	3492
/api/geocoding/reverse	580	773

Table 13: Average latency of 10 tests sending 232 concurrent requests

The average latency may be slightly lower than its 200-concurrent-request counterpart but that's because the previous test has inflated values due to the horrendous first test reaching 9700ms response time for the searching endpoint. The relative average is ~350ms higher excluding the spike.

One more observation that we can extract from the results is that the bigger the response body, in bytes, the slower will be the response. This isn't anything new or ground breaking but considering our application returns a variable number of results, it could play a factor significant or otherwise.

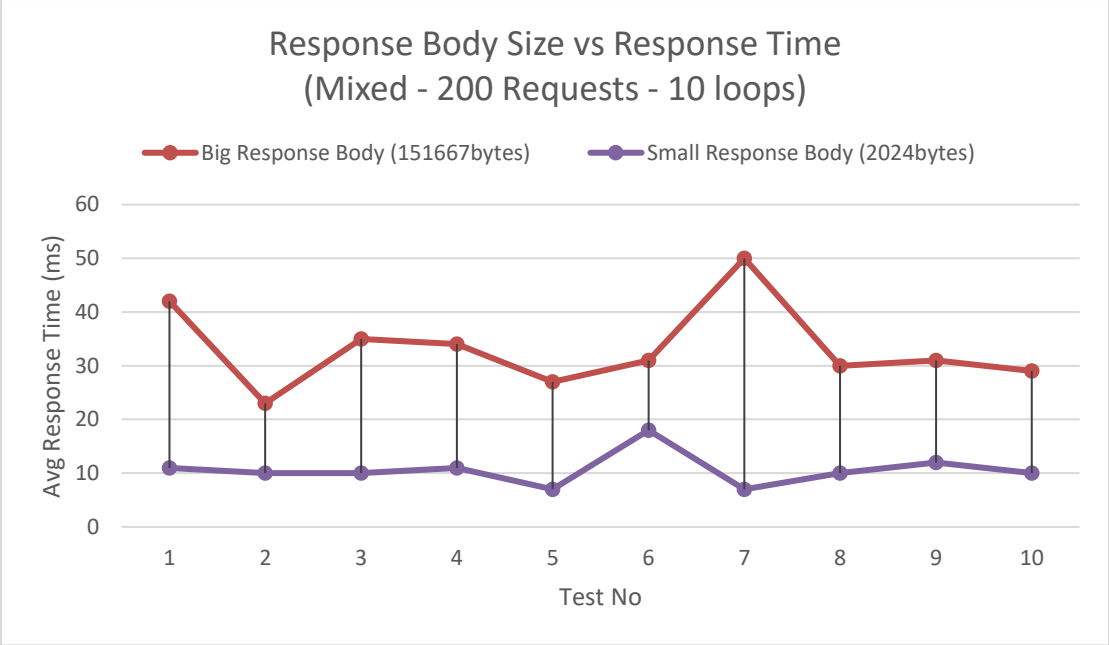
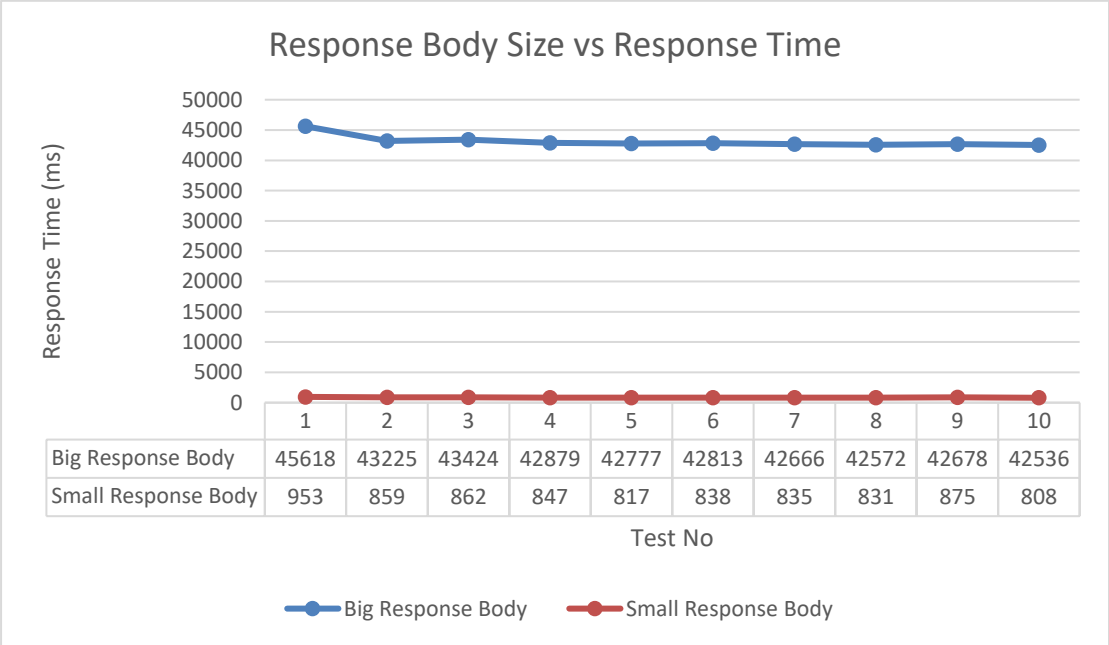


Table 14: Avg Response Time for Big vs Small Response body

While miniscule we will try to see if it will have a significant impact on the searching endpoint by comparing a request which returns a small amount of results vs a request which entails a bigger response body. For its comparable counterpart we will use a request which with the current dataset store inside the database returns 507 results around the New York area. We won't be sending multiple concurrent requests as that would pollute our finding by applying extra factors to the average response time.



	1	2	3	4	5	6	7	8	9	10
Big Response Body	45618	43225	43424	42879	42777	42813	42666	42572	42678	42536
Small Response Body	953	859	862	847	817	838	835	831	875	808

Table 15: Response Time for variable Response body size

The results can be considered conclusive in the fact that the *less* restrictive a user query is, the more it will *probably* take to be completed. By less restrictive, we mean no filtering the results before being transmitted and huge point of interest coverage. The big response body above was caused by a query containing the following point of interest:

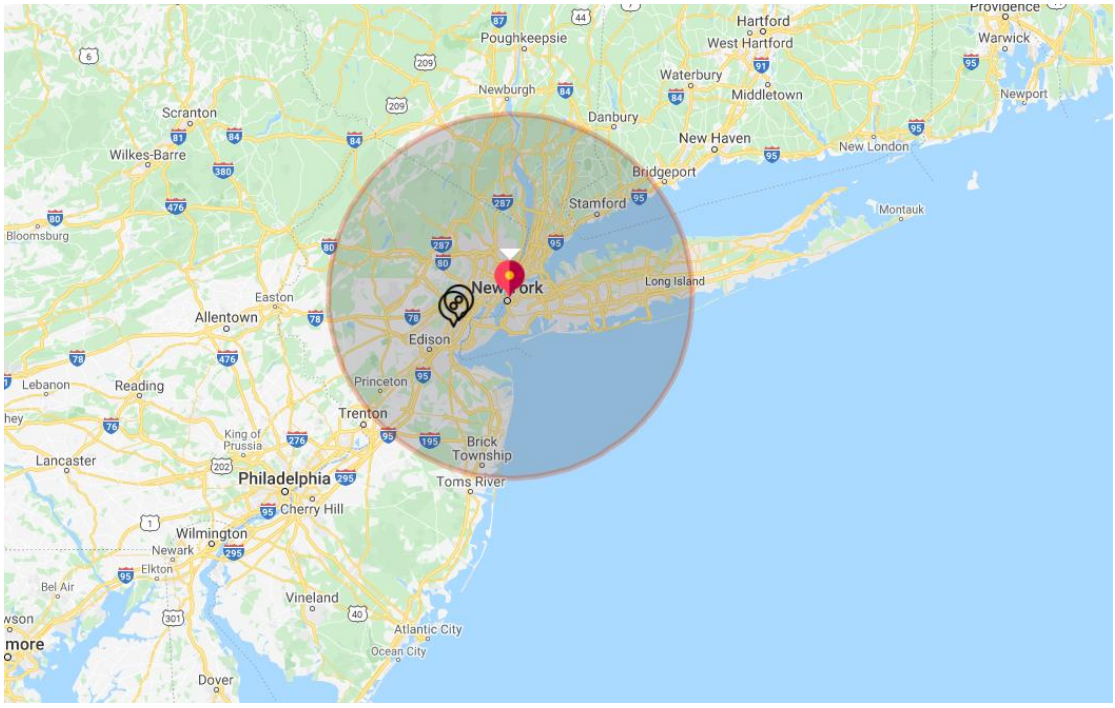


Image 21: Coverage by a single point of interest

*All the data from the performed tests are included in the spreadsheets accompanying this paper.

Chapter 5. Epilogue

5.1 Conclusion

Modern day businesses which are trying to achieve maximum throughput rely heavily on fast and efficient logistics. Resupplying and warehousing stock play a key part in staying competitive in today's market. We created an application that caters to those businesses by giving them a way to find commercial space for all purposes and needs.

In essence, this project was about developing a full stack application that would be able to accept and process a custom query and retrieve all the data that satisfied that particular query. The custom query would be a way for users to search for commercial real-estate, that are stored within our database, by specifying a region of their choice where they either own an establishment in that region or just looking to expand their business.

The tool developed is a very specialized one in the sense of there not being another similar tool in the realm of real-estate search engines that offers the same functionality. All the available vendors either offer a subset of the overall functionality by providing searching capabilities around a specific point but not more than one or they don't offer it at all.

This project was intended to provide a general direction and insight into developing a full stack web application by presenting our design choices, methodology, testing and evaluation. We managed to create an application that is capable of searching our database with the given constraints and doing so in a concurrent way meaning that it is capable of a decent sized scalability.

5.2 Future work

We think that a possible extension could be the implementation a more complex spatial pattern business model that is going to use our application as a foundation to be built upon. These patterns could be used to improve the search functionality of the application and to decrease the response time of it. In more detail, since our developed application is concerned with commercial properties and intended to be used by businesses, it could be interesting to create a way for a business owner to input their whole network of stores and be able to define resupplying route and budgetary constraints that give him the upper edge.

On a technical extension, we would propose implementing a role-based authentication system on the backend of our application that is used, among other things, to protect our API's endpoints from unauthorized usage to prevent excess charges from 3rd party vendors.

Bibliography

[1] *IEEE Recommended Practice for Software Requirements Specifications*, IEEE Std 830-1998, 1998.

[RaGe03] Raghu Ramakrishnan, Johannes Gehrke, 2003, Introduction to Database Design, *Database Management Systems.*

[2] Testing the Web Layer, *Spring Boot documentation.*

[LLE97] Leutenegger, Scott & Lopez, Mario & Edgington, Jeffrey. (1997). *STR: A Simple and Efficient Algorithm for R-Tree Packing*. Proc. VLDB Conf. 497-506. 10.1109/ICDE.1997.582015.

[3] *React JS documentation*