Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Рекомендаційна система пошуку житла базована на ізохронних мапах

Текстова частина
магістерської роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник магістерської роботи
д.т.н., доц. А. М. Глибовець

_____
(підпис)

“____” _____ 2021 р.

Виконав студент М. А. Жук

“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики, к.ф.-м.н.
_____С. С. Гороховський
(підпис)
„____”_____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2 р.н. магістерської програми Інженерія програмного забезпечення
Жуку Максиму Анатолійовичу

Розробити Архітектуру рекомендаційної системи пошуку житла, що базується на ізохронних мапах

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1 Огляд ключових компонентів рекомендаційної системи
2 Розробка архітектури рекомендаційної системи
3 Реалізація прототипу на основі отриманих знань та розробленої архітектури.
Висновки
Список літератури
Додатки

Дата видачі „___” _____ 2020 р.
Керівник А.М. Глибовець, доктор технічних наук, доцент _____(підпис)
Завдання отримав М. А. Жук _____(підпис)

Тема: Архітектуру рекомендаційної системи пошуку житла, що базується на ізохронних мапах

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	01.11.2020	
2.	Огляд технічної літератури за темою роботи	15.11.2020	
3.	Аналіз доменної області та сучасних рішень	29.11.2020	
	Аналіз ключових компонентів системи	27.12.2020	
3.	Розробка архітектури рекомендаційної системи	17.01.2021	
4.	Реалізація прототипу	14.02.2021	
7.	Написання пояснювальної записки	24.04.2021	
8.	Створення слайдів для доповіді та написання доповіді	27.04.2021	
9.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи	30.04.2021	
10.	Коригування роботи за результатами попереднього захисту	5.05.2021	
11.	Остаточне оформлення пояснювальної записки та слайдів	10.05.2021	
12.	Захист магістерської роботи (проекту)	18.06.2021	

Студент **Жук М. А**

Керівник Глибовець А.М. **“____”_____**

Зміст

Анотація

В рамках даної роботи проведено огляд сучасних технологій, що можуть бути використані для побудови рекомендаційних систем з використанням мап та ізохронів. Розглянуто основні складнощі пов’язані з побудовою рекомендаційної системи пошуку житла. Розроблена архітектура рекомендаційної системи. Реалізовано прототип, що демонструє життєздатність даних підходів та дає змогу оцінити корисність для користувача.

Ключові слова: рекомендаційна система, ізохрон, пошук житла, Mapbox, Google Maps, Valhalla, Open Street Map, Open Source Routing Machine.

Вступ

Актуальність теми, наукове і практичне значення

Для платформ пошуку нерухомості бути попереду конкурентів - завдання не з простих, але пошук підходящої квартири мало змінився за останні 10 років. Виберіть ціну, площу, кількість кімнат,..? Ці стандартні запитання не відображають реальної складності життя. Найскладніше і найцінніше запитання - де? Однушка в центрі Києва буде коштувати у кілька разів дорожче ніж 3-кімнатна квартира на окраїні Кропивницького. Так само в різних локаціях Києва ціна може відрізнятись у кілька разів. Саме за місце розташування люди готові платити набагато більші гроші за своє житло. За результатами опитування DOM.RIA місцерозташування один із основних параметрів вибору житла [9]. Багато молодих людей готові жити у менших апартаментах, але з кращим розташуванням, сервісом, інфраструктурою. Такі апартаменти часто називають smart house. Світова доля ринку смарт апартаментів за прогнозами має зрости з 79 мільярдів доларів до 314 мільярдів доларів за наступні 6 років [19].

На жаль платформи пошуку житла в Україні стосовно цього не можуть запропонувати нічого кращого ніж банальний вибір міста чи району. Тому користувачі мають подумки уявляти, що з цього району, чи від цієї конкретної квартири я буду добиратись до роботи приблизно стільки часу, а з іншого стільки часу. Ці роздуми ускладнюються в рази, якщо квартира шукається для молодої сім'ї з 2-3 людей. Або якщо у твоєму житті є ще якісь інтереси крім роботи, наприклад улюблений тренажерний зал чи місце навчання. Такі роздуми вимагають багато зусиль та часу, вони неточні, що може призвести до того, що на житло буде витрачена більша кількість коштів ніж необхідно, а результат буде не той який очікували. Було б набагато зручніше, якби платформа пошуку житла орієнтувалась, на кінцевий результат - оптимізацію пересування між точками інтересу. Така інтелектуальна платформа пошуку житла зможе дати новий користувацький досвід і знайти неочевидні варіанти житла, які б сам користувач

скоріш за все не знайшов. Адже дуже важко врахувати всі особливості доріг, видів транспорту і т. п. просто уявляючи це в голові.

Постановка задачі

Розробити архітектуру та прототип інтелектуальної платформи пошуку житла в Україні, основану на часових відстанях від об’єктів інтересу. Дослідити доцільність використання ізохронів для візуалізації доступності об'єктів інтересу.

Об’єкт дослідження

Процес пошуку житла за допомогою онлайн платформ.

Предмет дослідження

Покращення процесу пошуку підходящого житла за допомогою візуальних підказок часової доступності до точок інтересу користувача.

Мета та конкретні завдання дослідження

Дослідити можливість створення платформи, що при пошуку житла орієнтується в першу чергу не на розміщення апартаментів у певному районі, а на доступність об’єктів інфраструктури, які цікавлять користувача. Розробити прототип. Оцінити економічну доцільність, та практичну цінність такої системи.

Для того, щоб конкретизувати завдання, наведемо кілька практичних сценаріїв, на які будемо орієнтуватись, при розробці системи.

- Молода сучасна сім’я: чоловік, дружина, хлопчик шкільного віку. В сім’ї є одне авто, а хлопчик любить кататись на велосипеді. Сім’я хоче переїхати у нове житло, тому що поточне має не дуже зручне розташування і всі члени сім’ї витрачають багато часу на дорогу. Потрібно знайти таке житло, щоб хлопчик міг від нього за 15 хв доїхати до школи, а чоловік зміг менш ніж за 40 хв доїхати до роботи, при цьому підвезти дружину до її місця роботи. Проблема у тому, що

навіть не зважаючи, що всі члени вже давно живуть у місті, при пошуку житла важко тримати побажання всіх членів у голові і при цьому давати більш-менш точні часові оцінки.

- Молодий чоловік переїхав у столицю на нове місце роботи. Має машину, хоче добратися до нового місця роботи на машині не довше 20 хв. Проблема у тому, що він не достатньо добре знає місто і тим більше завантаженість трафіком конкретних доріг.
- Дівчина з провінції поступила в університет. Хоче знімати житло недалеко від університету, 15 - 20 хв пішки. Незважаючи на те, що вимога здається досить простою - конфігурація дворів і пішохідних доріжок, а також світлофорів, може бути досить непередбачуваною. Погано знає місто, тому важко оцінити часову доступність.

Джерела, огляд літератури

Термін ізохрон походить від двох грецьких слів ísos - рівний, та khrónos - час. Ізохронна мапа - це мапа, на якій візуально зображено ділянки до яких можна дістатись за певний проміжок часу з певної базової точки. Перші відомі використання ізохронних мап датуються ще 1881 [11]. Френсіс Гальтон, відомий англійський науковець, так само як і його двоюрідний брат, не менш відомий Чарльз Дарвін, мав тягу до подорожей. В 1881 Френсіс Гальтон опублікував карту доступності майже всіх куточків нашої планети з Лондона.

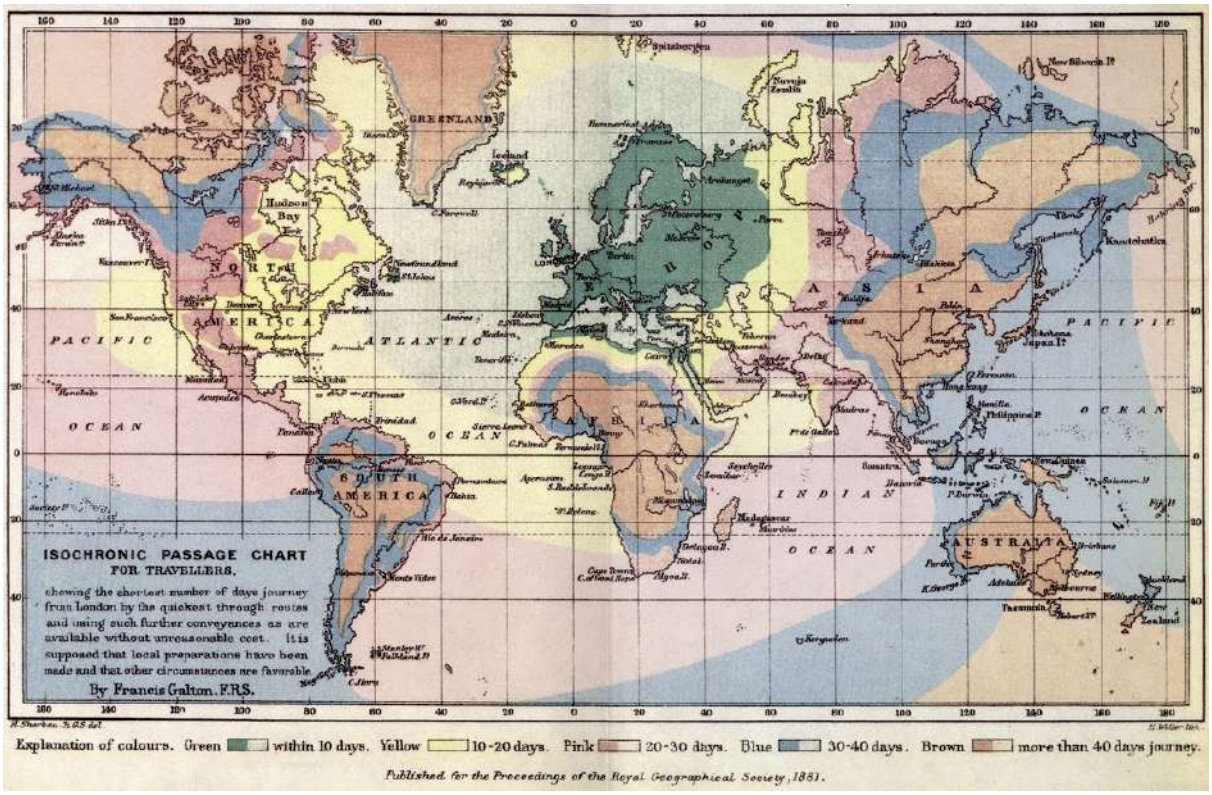


Рисунок. Публікація ізохрона Френсіса Гальтона.

Ізохронна мапа перш за все візуальний інструмент, який значно спрощує когнітивний процес і допомагає приймати більш складні рішення. Незважаючи на те, що ізохронні мапи дуже наглядні і зрозумілі для кінцевого споживача, вони не були надто поширені. На думку автора(на мою думку) пов’язано це з кількома причинами. Якщо розглянути мапу Гальтона, то одразу зрозуміло, що цінність цієї мапи сильно знижується, якщо ви не проживаєте в Лондоні або десь поблизу. Тому що ізохрон розраховується відносно базової точки. Також на мапу впливає багато факторів, які можуть достатньо швидко змінюватись з часом. Додалась нова залізнична колія, чи новий маршрут через океан і т. п. Також час буде сильно залежати від способу пересування: авто, пішки, велосипед... Крім того сам процес побудови мапи досить складний. Для побудови ізохронних мап використовуються алгоритми пошуку найкоротшого шляху у графах. Алгоритм Дейкстри [7] часто використовується для вирішення таких задач. Він Знаходить найкоротший шлях від однієї вершини графа до всіх інших вершин. Найкоротший шлях - це умовне

позначення, насправді це можуть бути будь-які ваги - наприклад час, як у нашому випадку. Найпростіша реалізація алгоритму Дейкстри потребує

O(V * V) дій, де V - це кількість вершин у графі. Враховуючи масштаби планети Земля і технологічний розвиток, для 1881 року це була досить складна задача. Проте алгоритми покращуються, з'являються нові технології, обчислювальні потужності збільшуються. За "законом" Мура обчислювальна потужність зростає в 2 раза кожні 2 роки. Пошук найкоротшого шляху відноситься до класу поліноміальних задач, тому ми можемо шукати рішення навіть на великих об'ємах даних. Тим більше, що для побудови ізохрона не обов'язково шукати найкоротший шлях до всіх вершин графа, так як ізохрони по своєму визначенню обмежені певним часовим проміжком, то якщо ми знайшли вершину графа дістатись, до якої потрібно більше часу ніж задано по умові, то далі рухатись вже немає сенсу.

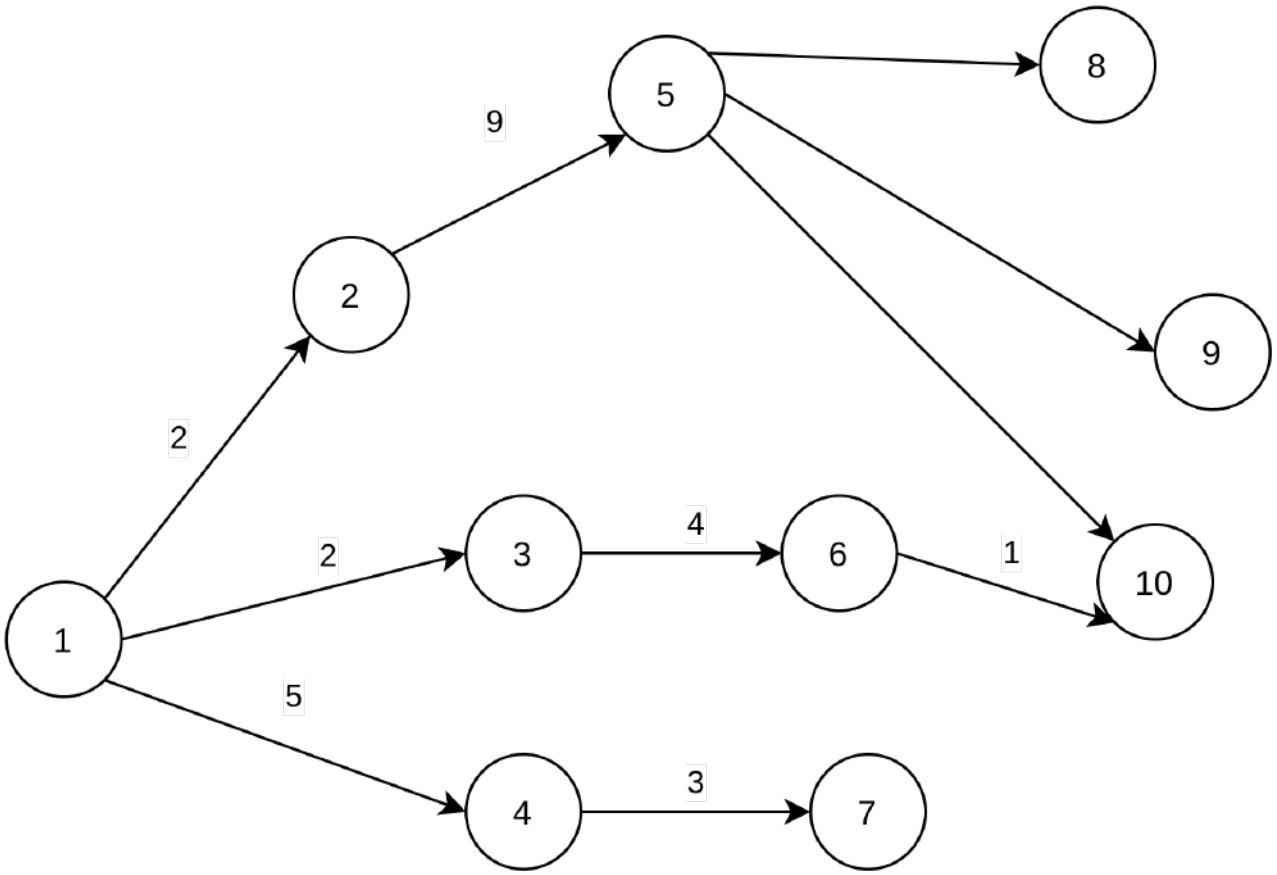


Рисунок. Граф для побудови ізохрону.

Припустимо, що ми будуємо ізохрон з графа зображеного на рисунку. Межа ізохрону - 10. З рисунку видно, що немає сенсу шукати шлях до вузла 5 проходить через вузол 2 і сумарно складає 11. Так це більше ніж границя нашого ізохрону, то далі немає сенсу шукати з вузла 5 шляхи до якихось інших вузлів. Тому вузли 8 і 9 просто випадають з нашого графу. А чим менший граф - тим швидше загалом відпрацює алгоритм.

Всі передостанні вершини нашого дерева найкоротших шялхів є межами нашого ізохрону. Маючи їх координати - можна легко нанести їх на мапу. Це й буде найпростіша реалізація ізохрону.

Проте такий спосіб виведення даних не самий інформативний. Для кращого відображення використовують інші методи, наприклад наприклад опуклу оболонку(Convex hull). В обчислювальній геометрії, прийнято використовувати термін «опукла оболонка» для межі мінімальної опуклої множини, що містить дану непорожню скінченну множину точок на площині. Для скінченної множини точок, опукла оболонка являє собою ламану лінію [8].

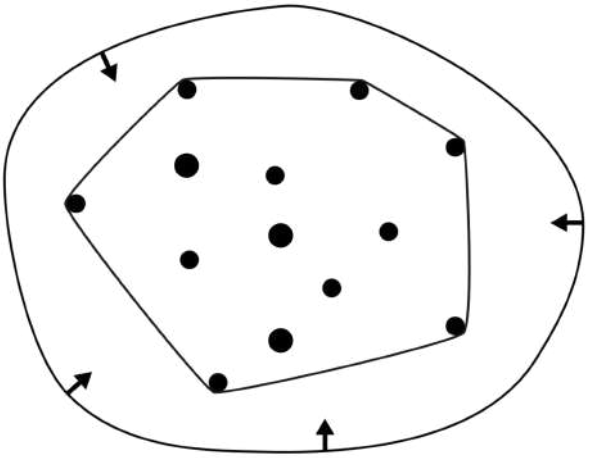


Рисунок. Опукла оболонка.

В опуклої оболонки є певні недоліки. В неї можуть потрапити зони, що є недосяжними. Наприклад, якщо у місті є метро, то доїхати від станції до станції можна за відносно короткий проміжок часу, але ми не можемо зійти на півдорозі, томі дістатись до об’єктів, що знаходяться між станціями не завжди можливо у

вказаний часовий діапазон. Тому ці об’єкти мають бути виключеними з опуклої оболонки. Більш точний метод - Альфа форми(Alpha shapes) [10]. Альфа форми в залежності від величини альфа можуть по різному огинати множину точок і давати полігони різної форми, які в тому числі допускають пустоти.

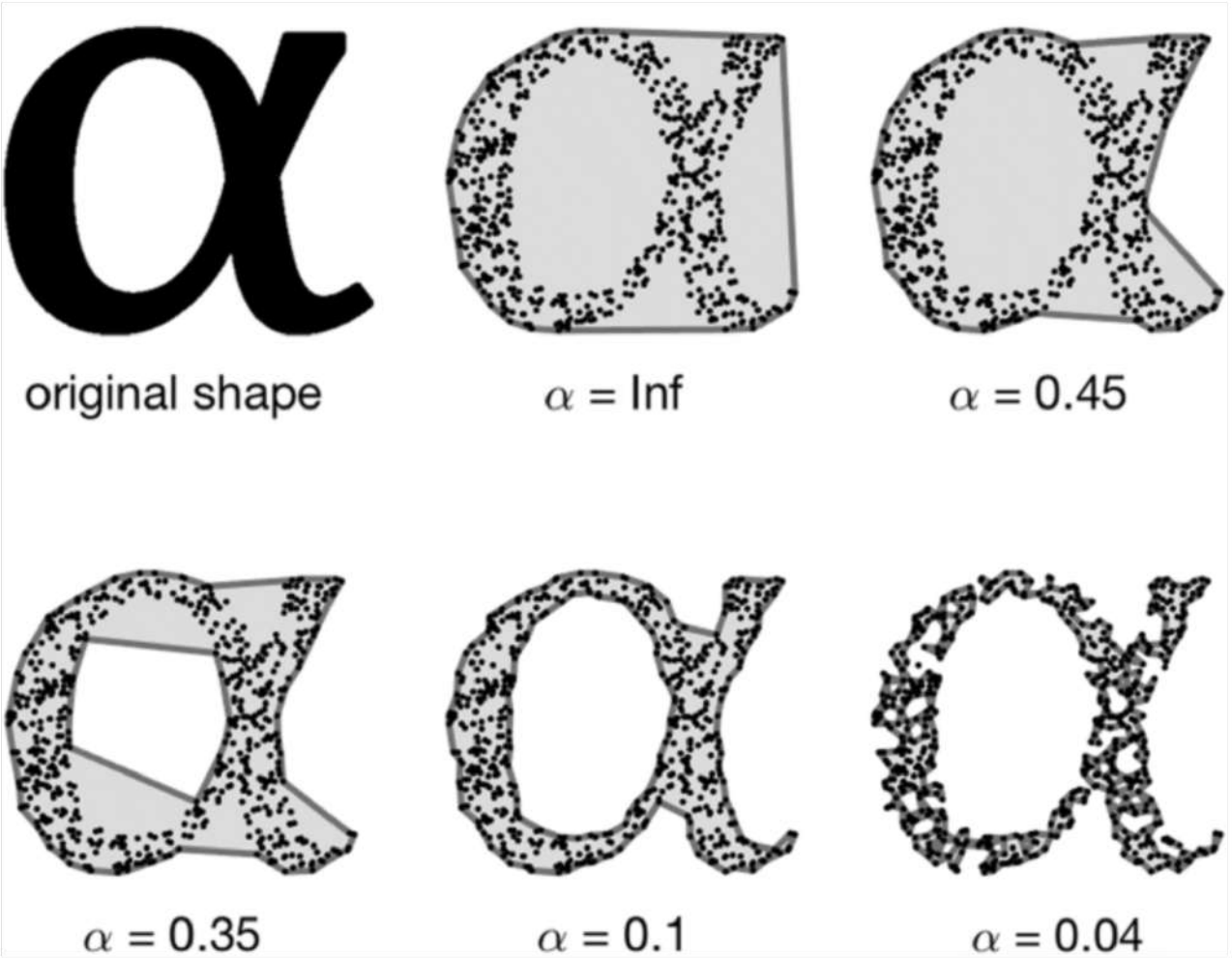


Рисунок. Різні альфа форм в залежності від альфа

Є інші методи створення контуру ізохрону, наприклад: метод 2D інтерполяції, метод заповнених контурів, або метод буферизвонаних доріг, розроблений спеціально для мап. Жоден з методів не є універсальним. Іноді їх комбінують. Також жоден з методів не являється абсолютно точним. Тобто в будь-якому разі контур ізохрона буде в певній мірі наближеною до істини

величиною. Але так як на час, щоб добратись з однієї точки до іншої впливає дуже багато факторів і це очевидно для користувача, то це є прийнятним рішенням.

Розділ 1. Огляд ключових компонентів рекомендаційної системи.

1.1 Вибір провайдера данних про житло

Для нашого прототипу нам потрібні дані про житло доступне для покупки чи оренди. Найбільші гравці на ринку України це DOM.RIA, LUN, OLX. OLX має лише API для роботи з оголошеннями конкретного користувача, за його згоди, для імпорту/експорту власних оголошень на інші платформи (“OLX Partner API”)[16]. LUN також не має API, так як його сервіс flatfy являється агрегатором і якщо вони дадуть легкий і вільний доступ до всіх об’єктів, то це знизить їх конкурентність на ринку. DOM.RIA єдиний портал із трійки лідерів, який має відкритий REST API з доступом до всіх об’єктів нерухомості [17]. Для доступу до API потрібно лише запросити унікальний ключ доступу. Таким чином ми отримуємо доступ до всіх об’єктів одного з найбільших порталів з нерухомості України. Крім того більшість об’єктів на DOM.RIA мають точну геолокацію, так як на формі додавання оголошення номер будинку є обов’язковим параметром.

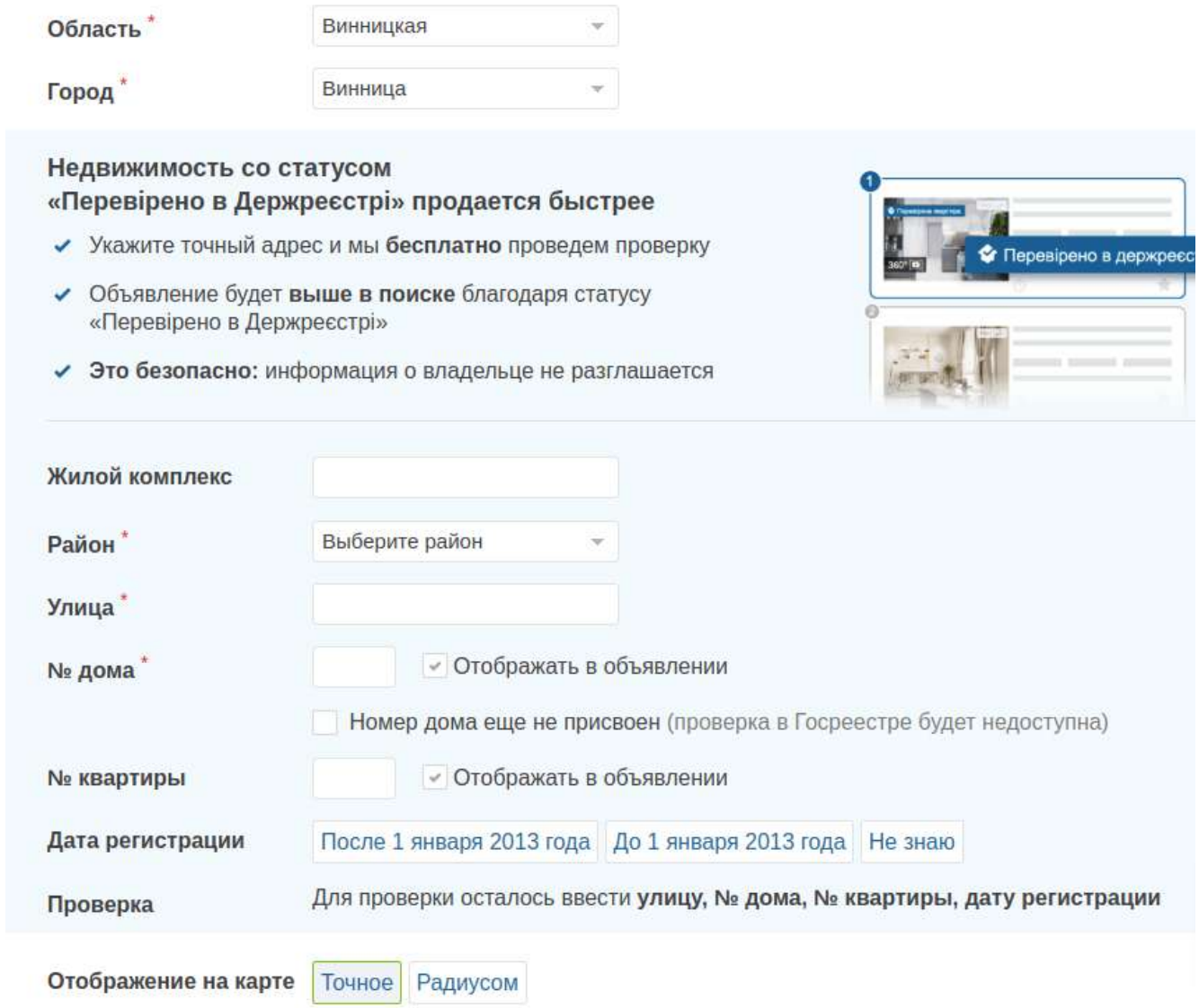


Рисунок 1.1.1. Гео параметри на формі додавання нового оголошення.

І хоч на сайті місце розташування виводиться у вигляді радіусу, але в самому оголошенні і API можна отримати точні координати.

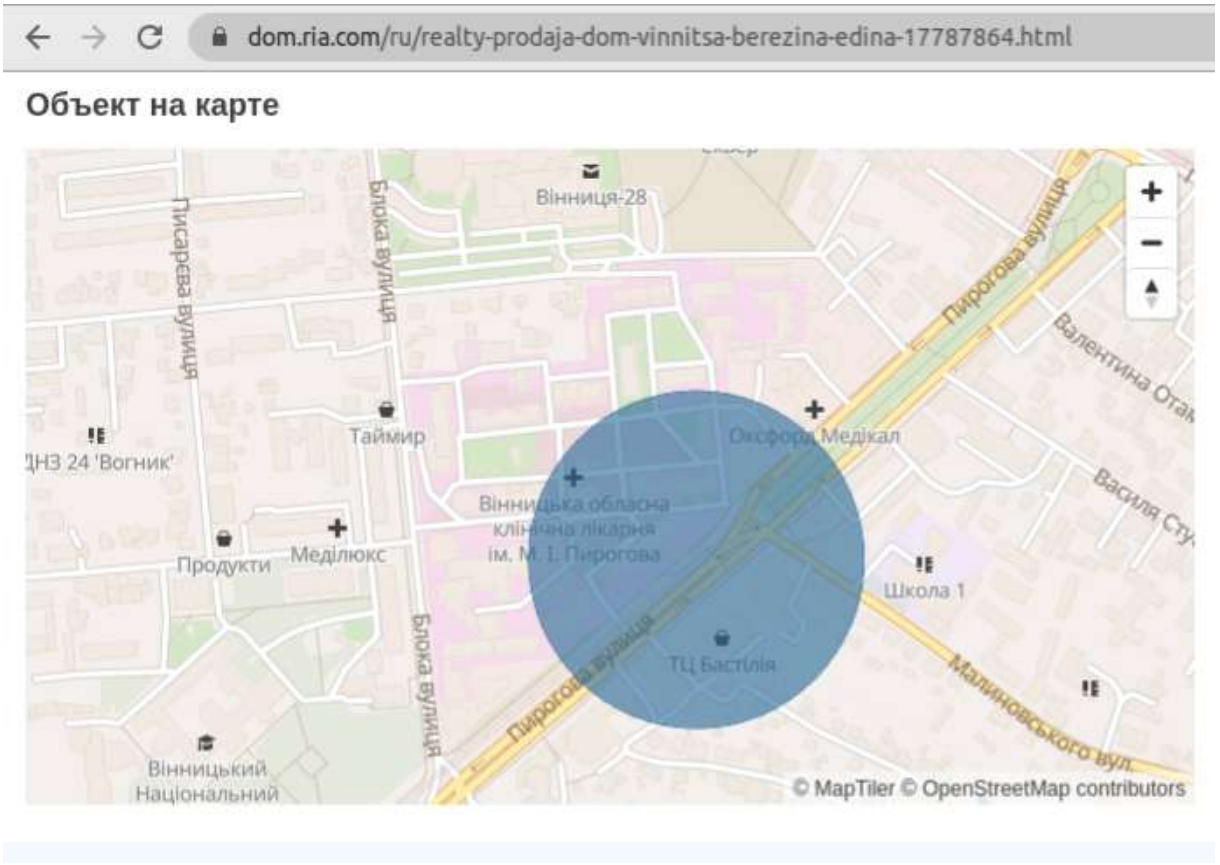


Рисунок 1.2. Зображення об’єкту нерухомості на мапі порталу DOM.RIA.

Рисунок 1.1.3. Дані, які надає відкритий API того ж самого об’єкту нерухомості.

Звичайно не добросовісний користувач може вказати будь-яку адресу, але для реалізації нашого прототипу можна знехтувати цим фактом. Крім того багато об’єктів перевіряються через державний реєстр, а також незалежним інспектором і ми можемо обмежитись лише перевіреними оголошеннями у разі необхідності

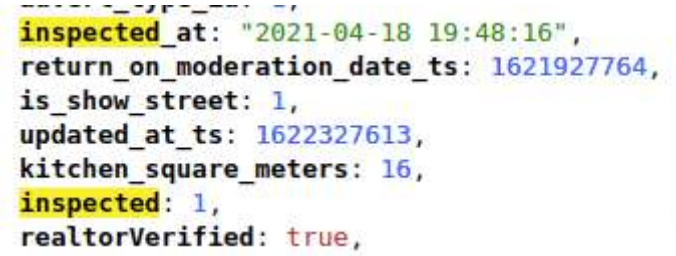


Рисунок 1.1.4. Об’єкт нерухомості з підтвердженим місцем розташування.

На момент момент написання роботи на платформі є більше 20 тисяч таких об’єктів, чого має бути цілком достатньо для прототипу.

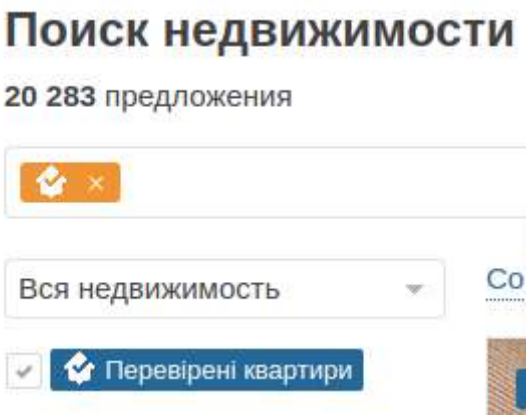


Рисунок 1.1.5. Загальна кількість об’єктів з перевіреною локацією.

1.2 Вибір провайдера мапи

1.2.1 Вибір платформи

Потенційно наш прототип сервісу пошуку житла можна зробити на будь-якій платформі, але мабуть найпростіше буде це зробити у вигляді веб застосунку. Браузер можна запустити на всіх популярних платформах, тому сервіс можна

вважати кросплатформенним. Також для браузерів є велика кількість клієнтських бібліотек для роботи з мапами і всі вони добре адаптуються під різні пристрої, а також часто мають зручний API для роботи з додатковими геосервісами, які нам можуть стати у нагоді. Більшість бібліотек безкоштовні, або мають певний безкоштовний ліміт використання самої мапи, а також додаткових геосервісів, чого має бути достатньо для прототипу. Для запуску повноцінного сервісу потрібно прораховувати витрати, та економічну доцільність, так як обрахунок ізохронів може бути досить ресурсоємною операцією і відповідно не дешевою на великих об’ємах вхідного трафіку на рекомендаційну систему пошуку житла.

1.2.2 Клієнтські бібліотеки

Після первинного аналізу виявилось що кількість клієнтських бібліотек присутніх на ринку вражаюча:

- Google Maps
- Yandex maps
- Leaflet
- Openlayers
- Datamaps
- Mapbox
- Bing Maps
- Jvectormap
- Amcharts Map Chart
- ArcGIS
- Kartograph
- Zeemaps
- Highmaps by Highcharts
- Anymap by Anychart
- Cesium

- Mapael
- Polymaps

Для того щоб звузити коло кандидатів скористаємось кількома прийомами. Простий і досить надійний спосіб зменшити коло - довіритись оцінкам інших розробників. Популярність бібліотеки говорить про те, що вона має хороший функціонал, є багато розробників, які з нею працюють, бібліотека буде підтримуватись ще довгий час, тому що має здорове community. Для цього ми скористаємось наступними ресурсами:

- npm - найбільший репозиторій бібліотек на javascript, налічує понад мільйон бібліотек
- google trends - аналітика пошукових запитів по всьому світі від google
- github - веб портал для розробників оснований на найпопулярнішій системі контролю версій - git. Має багато статистичних даних, та рейтингову систему.
- npm trends - ресурс, який збирає і наглядно відображає статистику бібліотек з npm. Дозволяє вибирати для порівняння кілька бібліотек, та різні часові періоди.

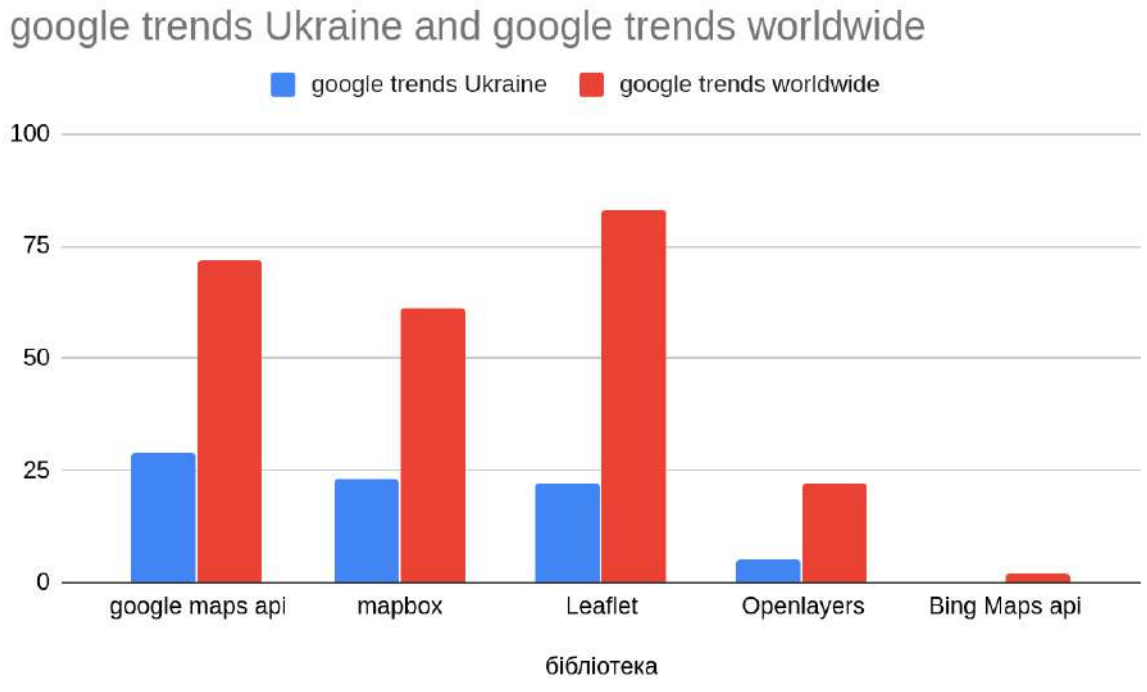


Рисунок 1.2.2.1. Порівняння бібліотек за google trends.

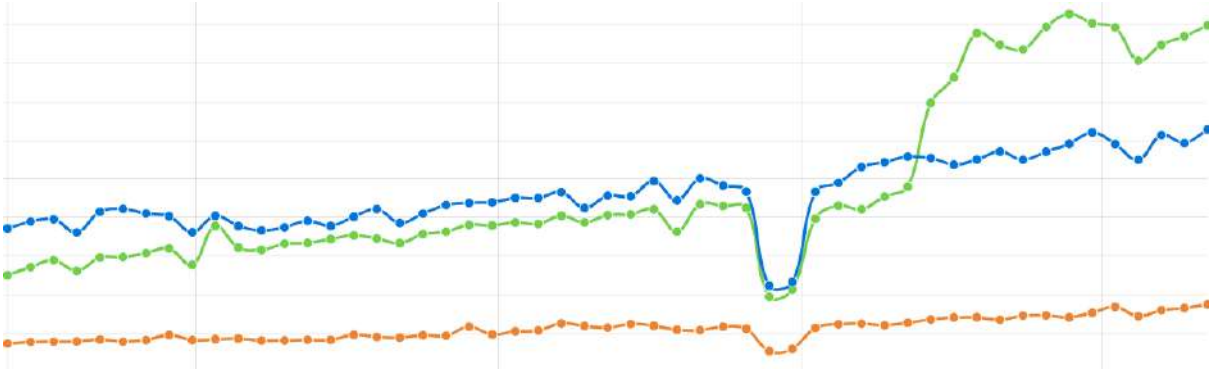
По результатам Google Trends найпопулярнішими як в Україні так і світі є Google maps, Mapbox та Leaflet. Важливо, що Google maps популярні не лише як платформа для розробки користувацьких карт, а і як окремий продукт. Це важливий фактор з точки зору зрозумілості інтерфейсів, адже багато користувачів звикло до інтерфейсів Google maps. Проте в Google maps є один суттєвий недолік, при великих об'ємах запитів вони мають найдорожчий API.

	MONTHLY VOLUME RANGE (Price per MAP LOAD)
0–100,000	100,001–500,000
0.007 USD per each (7.00 USD per 1000)	0.0056 USD per each (5.60 USD per 1000)

Рисунок 1.2.2.2. Ціни на базовий модуль Google Maps JavaScript API

За оцінками github Leaflet займає лідуючу позицію і має на момент написання роботи майже 31 тисячу зірок від користувачів. Найближчі конкуренти Mapbox та Openlayers мають 7 та 8 тисяч відповідно. На жаль Google Maps не open source рішення, тому ми не можемо його долучити до цього порівняння.

Mapbox завантажують з npm майже 500 тисяч разів за тиждень, а Leaflet лише 400 тисяч. Це не велика різниця. Leaflet має близько 300 відкритих питань на github, а Mapbox аж у 2 рази більше. Але зважаючи на те, що Mapbox має набагато більшу функціональність - більша кількість відкритих питань може бути цілком виправдана. На момент написання Leaflet останній раз оновлювався 9 місяців тому. Це не найкращий показник. Тоді як Mapbox оновлювався лише 5 днів тому. Mapbox підтримує у 2 рази більше розробників - 12 проти 5 у Leaflet. Тому цілком логічно, що бібліотека частіше оновлюється і потенційно більш стабільна. Нажаль Google Maps не використовує npm, тому нам не вдасться їх порівняти.



Графік 1.2.2.3. Кількість завантажень за останній рік(npmtrends.com).

	Openlayers	Mapbox	Leaflet
Github stars	7 000	6 000	28 000
npm downloads per week	87 000	400 000	300 000
issues	102	600	395
last update	3 days	5 days	5 month
contributors	3	12	5
size	6.5 MB	30.5 MB	3.4 MB

Таблиця 1.2.2.4 - порівняльна таблиця клієнтських бібліотек.

1.2.3 Підложка мапи

З технічної точки зору мапа представляє собою деяку підложку, де зображені основні географічні об’єкти: рельєф, дороги, будинки. Поверх цієї підложки ми можемо нанести ще шари додаткових об’єктів, які нам потрібні. Таким чином мапа являє собою багатошарову структуру різного рівня прозорості. Причому ці шари технічно можуть бути виконані зовсім по різному, якщо це дозволяє бібліотека мапи.

Перша мапа в інтернеті була показана компанією Xerox ще у 1993 році [18]. Вона являла собою одну велику картинку і при кожній взаємодії користувача з нею потрібно було завантажувати нову картинку, для данного конкретно екрану, широти, довготи, масштабу. Швидкодія мапи була далекою від ідеалу, особливо, якщо врахувати швидкість інтернету в ті роки, а користувачам потрібно було дуже

довго очікувати після кожної взаємодії, поки завантажується нова картинка. Але навіть така мапа була значним проривом у порівнянні з традиційними паперовими мапами.

В 2005 році компанія Google запропонувала новий підхід, який створив революцією на ринку мап і дефакто стандартом вже більше 15 років. Цей підхід дуже схожий на те, як взаємодіють з великими паперовими мапами. Великі мапи зазвичай продаються зігнутими у кілька разів. Так їх набагато зручніше переносити і переглядати, адже не завжди є можливість розкласти мапу повністю. Схожий принцип використовується і в онлайн мапах. Вся мапа розділена на заздалегідь визначені рівні прямокутники - плитки. Так як розміри відомі, то їх можна заздалегідь згенерувати [20] для кожного рівня наближення мапи. Ви завантажуєте лише ті плитки, які вам необхідні, а при зміщенні мапи дозавантажуєте ще кілька плиток, яких вам не вистачає. При збільшенні мапи, можна спочатку розтягнути наявні зображення, а по мірі надходження більш точних зображень, змінювати ці області. Те саме можна робити і при зменшенні. При нестабільному каналі інтернету можна іноді помітити, як дозавантажуються окремі плитки, або притягуються, а потім стають більш чіткими.

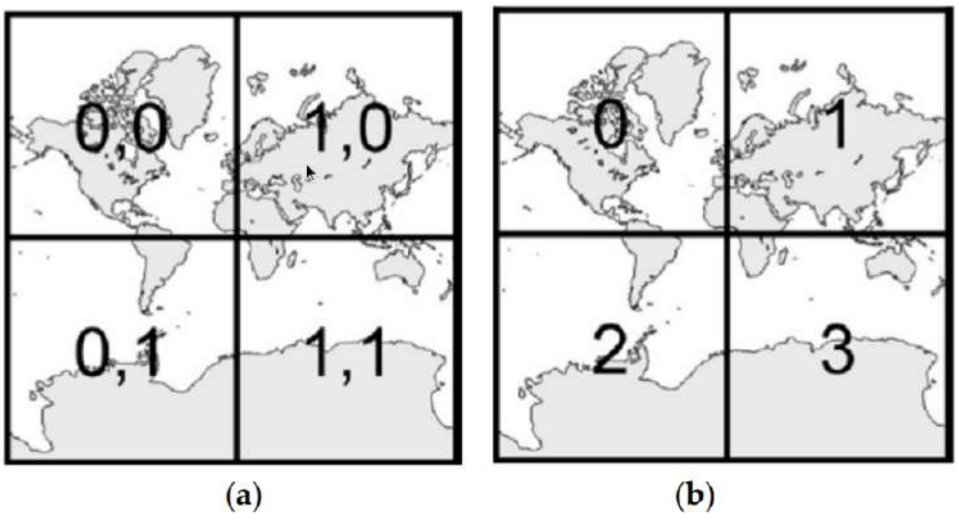


Рисунок 1.2.3.1 - Схема плиток Google(a) та Bing(b).

Є два типи плиток: векторні і растрові. По суті це два типи зображень: векторне і растрове. Растрові кодуються, як набір точок, а векторні, як набір векторів.

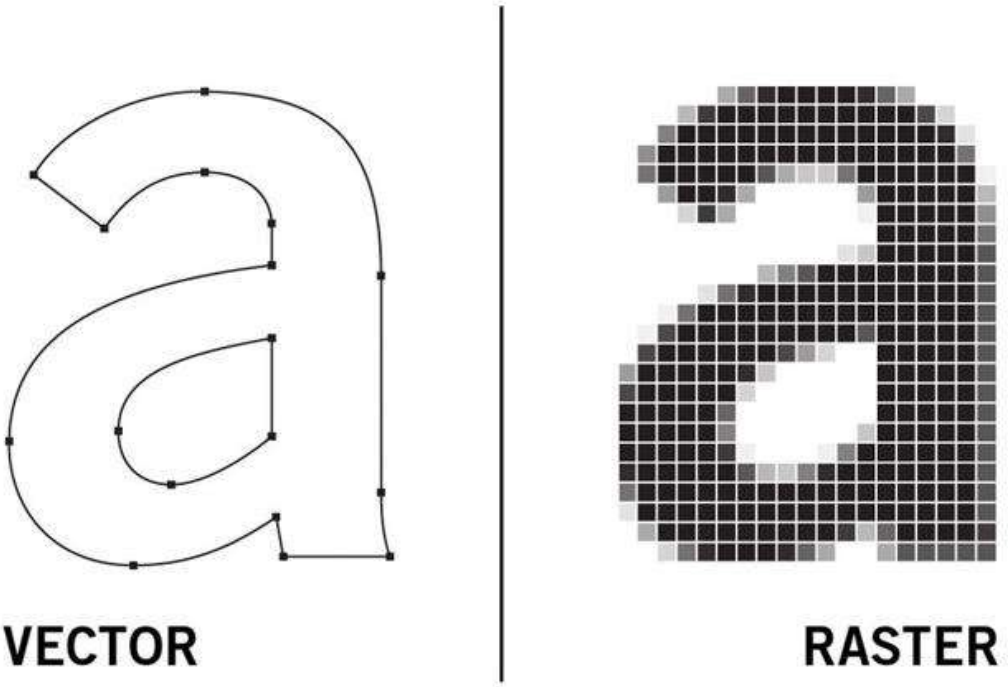


Рисунок 1.2.3.2. - Різниця між растровими і векторними зображеннями.

І в тих і в інших є свої переваги і недоліки. Растрових плитки мають високу швидкість генерації у порівнянні з векторними. Їх можливо навіть створювати в майже реальному часі, за кілька секунд, і таким чином можна використовувати менше дискового простору. Також растрові плитки підтримуються практично всіма браузерами. Але є в них і деякі недоліки. При будь-якій зміні на карті їх потрібно повністю генерувати заново. Також з растровою мапою важко реалізувати інтерактивну взаємодію з користувачем [4]. Пов’язано це з тим, що з боку браузера це лише зображення і відокремити на цьому окремі об’єкти для реалізації інтерактивності - досить нетривіальна задача.

У 2010 році компанією Google був запропонований новий підхід - векторні плитки [3]. Суть полягала у тому, щоб генерувати на сервері не зображення, а опис об’єктів на мапі: точки, лінії, полігони. А також опис до об’єктів: назви міст, селищ, об’єктів інфраструктури. Дані зазвичай зберігаються у спеціальному

форматі - GeoJSON. Він оснований на форматі JSON, який із-за своєї простоти наглядності і компактності у порівнянні з xml являється зараз найпопулярнішим форматом передачі даних. Ще одним популярним спеціалізованим форматом є Mapbox Vector Tile. MVT використовує protobuf - бінарний протокол, а тому більш компактний ніж GeoJson. Однією з ключових переваг векторних плиток є здатність динамічно змінювати візуальну стилізацію мапи. Це реалізовано за рахунок того, що об'єкти, що зображуються на мапі та візуальні стилі цих об'єктів зберігаються у різних файлах.

Варто зазначити, що незважаючи на зазначені вище переваги векторних плиток Google і досі часто використовує растрові плитки. Скоріш за все це пов'язано з тим, що і ті і інші мають свої переваги та недоліки і в залежності від виду мапи - вибирається відповідний тип підложки.

1.2.4 Вибір клієнтської бібліотеки

Після першої фази відсіювання, у нас залишилось 3 бібліотеки. Google Maps, Leaflet та Mapbox. Google Maps ми також відкинемо, так як це рішення повністю пропрієтарне і досить дороге для ринку України.

Залишається дві бібліотеки Mapbox та Leaflet. Обидві базуються на даних з OSM(open street map). Також обидів opensource рішення.

Leaflet має досить обмежену підтримку векторних плиток. Є один плагін Leaflet.MapboxVectorTile, в якому реалізований рендерінг векторних плиток за допомогою canvas. Проте це важко назвати надійним рішенням з точки зору Bus Factor. Mapbox вміє рендерити картинку з векторних даних використовуючи графічне ядро. Це вимагає підтримки openGL, але дає дуже хорошу швидкодію. Для сучасних браузерів це не є проблемою [2]. Також варто відмітити, що браузер Chrome має деякі оптимізації для рендеринга canvas, які використовують GPU. Загалом бібліотека Mapbox основана на Leaflet, про що написано у документації, на офіційному сайті [13]. Загалом Mapbox має кращу підтримку векторних плиток, ширший функціонал, кращу підтримку, а також надає доступ до API Leaflet. Тому

нашим вибором буде Mapbox, так як він має широкий та варіативний функціонал, хорошу підтримку, та являється opensource рішенням.

1.3 Кластеризація та фільтрація

Так як загальна кількість об’єктів досить велика, може сягати сотні тисяч, то варто задуматись про аспект швидкодії. Адже це не легка задача - одночасно вивести десятки тисяч об’єктів нерухомості. Також потрібно ці дані передавати з сервера у браузер. Якщо передавати всі об’єкти, то це може бути кілька мегабайт, що досить суттєво для мобільних пристроїв. Крім того, якщо реалізовувати фільтрацію у нашій пошуковій системі, за певними параметрами, то при виборі користувачем якогось фільтру, потрібно буде перезавантажувати ці дані знов і знов.

Це також незручно з точки зору користувача, тому що при такій кількості об’єктів мапа перетвориться на неоднорідну плямисту масу одного кольору, на якій буде важко орієнтуватись. Для того щоб покращити швидкодію та UX використовують кластеризацію.

Кластеризація - це процес групування близько розташованих об’єктів. Є два способи кластеризації. Можливо просто відсікати частину точок, щоб не створювати незрозумілі плями на мапі, можна об’єднувати групу близько розташованих точок у кластерний елемент і відображати у ньому кількість точок, що входить у нього. Сам процес кластеризації можна проводити як у браузері, так і на сервері.

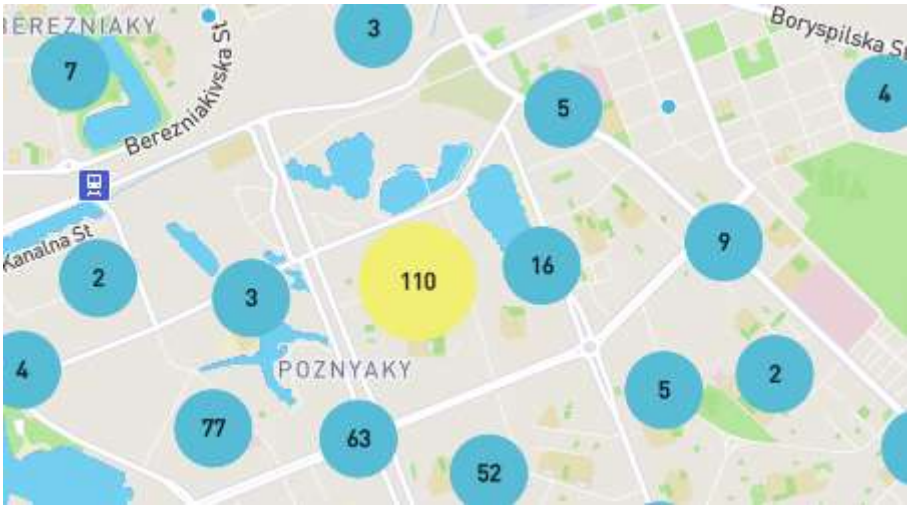


Рисунок 2.4.1 - Кластеризація.

Кластеризацію на клієнті простіше реалізувати ніж на сервері, так як на клієнті відомі характеристики вашої мапи і бібліотеки кластеризації часто інтегруються в бібліотеку роботи з мапою. Проте суттєвим недоліком є необхідність передавати всі дані на клієнт. Крім того, сам процес кластеризації вимагає обчислювальних ресурсів користувача. Тому якщо мапа має працювати на смартфонах, то краще не використовувати клієнтську кластеризацію на великих об’ємах даних.

Також при реалізації кластеризації слід звертати увагу на те чи потрібно дані фільтрувати. Тому що при поєднання фільтрації і серверної кластеризації ми не можемо наперед згенерувати кластери разом з нашими плитками, тому що генерація плиток займає суттєвий час, а заздалегідь згенерувати кластери не вдасться, тому що кількість об’єктів у кластері буде різна для різних наборів фільтрів. Якщо генерувати заздалегідь окремі плитки для кожної комбінації фільтрів - то це займе дуже багато часу і дуже багато дискового простору, при тому, що певні набори фільтрів користувачі можуть і не вибирати на практиці.

Для того щоб вирішити цю проблему можна або не показувати кількість об’єктів в кластері або генерувати кластери у реальному часі. Перший підхід звичайно простіший, але дає гірший користувацький досвід. При генерації плиток для мапи у областях з великим скупченням точок, деякі точки будуть відсікатись

на відповідних рівнях збільшення. Чим більше наближення - тим більше точок користувач бачить. Такий підхід використовує багато платформ, наприклад Booking.com. При цьому більшість користувачів навіть не зрозуміє, що точок було менше, тому що вони і так перекривали одна одну на маленькому збільшенні мапи. Фільтрація при такому підході може забезпечуватись за рахунок зміни стилю відображення точок. Окремий сервіс фільтрації може надавати інформацію, який набір точок має бути видимий, а mapbox за рахунок стилізації змінює їх зовнішній вигляд, щоб користувачу було зрозуміло, що він відфільтрував. Один із недоліків заключається у тому, що на маленькому збільшенні мапи не всі точки передаються з серверу, тому що ми їх кластеризували і частину відкинули. Тому при певних комбінаціях фільтрів деякі області на мапі будуть виглядати так, ніби там немає об’єктів, які нас цікавлять, проте це не так. Звичайно при збільшенні мапи, ми їх побачимо. Але навіщо нам збільшувати мапу у тому місці, де мапа не показує цікавих нам об’єктів?

Другий варіант більш складний у реалізації, але дає кращий користувацький досвід. Для цього створюється мапа з декількох шарів. Перший шар - це плитки з основною інформацією яку ми бачимо на мапі: вулиці, будинки, дороги. Другий шар - це об’єкти, до яких можуть бути примінені фільтри. Таким чином нам не потрібно генерувати “нальоту” всю мапу, а лише шар з об’єктами і кластерами у відповідності до вибраних користувачем фільтрів [14].

Кластери потім можуть бути передані як GeoJson, або конвертовані у векторні плитки. Geojson - більш простий спосіб, але як вже згадувалось раніше - векторні плитки використовують бінарний протокол і мають менший розмір, проте генерація векторних плиток займає більше часу. Тому для досягнення найкращих показників потрібно експериментувати з реальною аудиторією. У більшості випадків користувачі не відчують різниці між двома варіантами, так як ще на етапі кластеризації буде відкинуто основну кількість точок. Mapbox підтримує обидва формата, що дає нам змогу адаптуватись у разі необхідності.

1.4 Геокодування та автодоповнення.

Геокодування - це процес знаходження місця знаходження об’єкту на мапі за його назвою. Також є зворотне геокодування, коли за місцем розташування, ми можемо отримати список об’єктів на мапі. Сервіси геокодування часто підтримуються підтримують автодоповнення - це дає змогу побудувати зручний інтерфейс для користувача, коли він вводить перші літери об’єкту інфраструктури, а сервіс геокодування підказує йому повну назву, з можливістю вибору. Автодоповнення з технічної точки зору досить складна задача. Адже сервіс геокодування має надавати відповідь зі швидкістю яка співставляється зі швидкістю набору літер користувачем, а також витримувати велику кількість одночасних запитів, так як кожна введена користувачем літера - генерує новий запит на сервер. Mapbox надає безкоштовний доступ до власного сервісу геокодування, якщо ви генеруєте до 100 тисяч запитів за місяць і 0.75 $ за кожну наступну тисячу. Після 500 тисяч ціна зменшується до 0.6 $ за тисячу.

Варто також пам’ятати, що API геокодування mapbox нічого не знає про об’єкти нерухомості, які ми додаємо на мапу. Наприклад якщо користувач буде шукати квартиру у новобудові “Нове життя” - скоріше за все цієї новобудови ще не буде в базі openstreetmap і відповідно у mapbox. Тому для кращої реалізації сервісу геокодування потрібно доповнювати комбінувати дані з різних джерел. Також можливо повністю відмовитись від сервісу Mapbox і реалізувати геокодування в рамках рекомендаційної системи.

1.5 Алгоритми побудови ізохронів

Всі алгоритми побудови ізохронів, які використовуються в наш час і досить поширені, грубо можна поділити на дві групи: спеціалізовані алгоритми і наближені алгоритми.

Спеціалізовані алгоритми мають кращу швидкодію, більш ефективні у використанні ресурсів, більш точні, фінансово вигідніші. Але адаптація в існуючу

архітектуру не завжди фінансово виправдана, тому часто користуються наближеними алгоритмами.

1.5.1 Наближені алгоритми

Наближені алгоритми базуються на певних work-around, у зв’язку з тим, що певні сервіси не надають потрібного функціоналу. Але він може бути реалізований іншим чином і давати прийнятний результат. Наприклад ви уже побудували мапу на базі google maps, і ви не хочете ускладнювати архітектуру, додатковими вендорами, і вам не потрібна максимальна точність ізохрону. Google Maps не мають api для побудови ізохронів, але ви можете імітувати його, через distance matrix api [5]. Суть алгоритма наступна. Навколо вихідної точки малюється кілька уявних кіл. Ці кола мають проходити через потенційні краї ізохрона. Їх можна приблизно розрахувати виходячи з швидкості пересування: авто, пішки,.. Потім по цих колах рівномірно розподіляють точки, які ми будемо передавати на distance matrix api. Особливість цього API що йому можна передавати не лише дві точки: початкову і вихідну, а цілий набір точок. І відповідно отримувати цілий набір часових метрик для кожної пари точок. Такі API є у багатьох провайдерів: Google Maps, Mapbox, TomTom,.. Чим більше кіл буде і чим більше на них точок - тим більш точний буде ізохрон. Проте API як правило має ліміти, наприклад Google Maps приймає не більш ніж 25 пар. Тому для більшої точності необхідно робити декілька запитів, що робить кінцеве рішення дорожчим. Маючи часові відстані до ключових точок, можемо далі методом інтерполяції визначити будь-які проміжні точки. Тут у нагоді може стати бібліотека turf.js. Вона має дуже багато зручних функції для просторового аналізу. Ми можемо створити сітку точок, за допомогою функції pointGrid.

Потім за допомогою функції tin створюємо трикутну нерегулярну мережу. Трикутна нерегулярна мережа(Triangulated irregular network) являє собою суцільну поверхню, що складається повністю з трикутних граней. Ці трикутники(полігони) у свою чергу ми використаємо у функції planepoint, яка власне і виконує інтерполяцію. Таким чином ми можемо визначити часову дистанцію для всіх

точок у нашій згенерованій сітці точок. І далі за допомогою фільтрації необхідних нам точок за часовим критерієм малювати на мапі ізохрони.

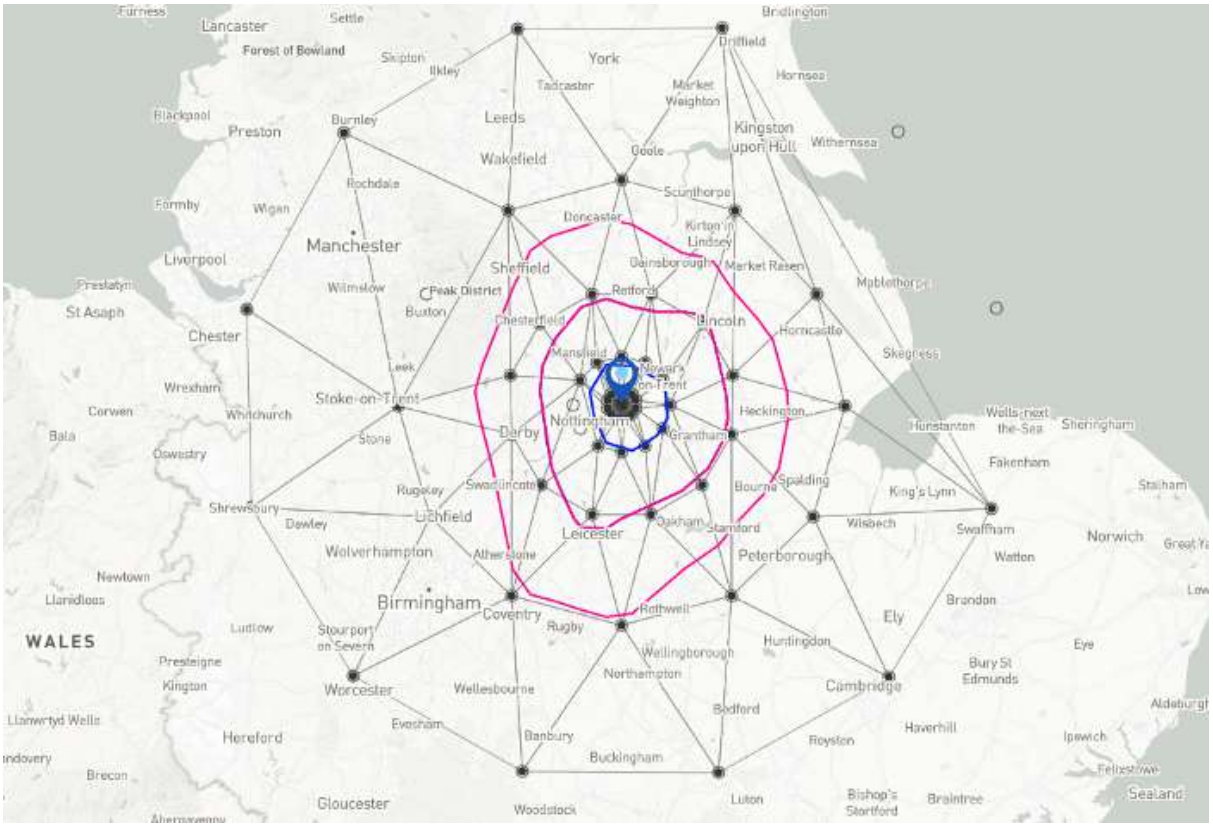


Рисунок 1.5.1.1. Опорні точки(чорні), трикутна нерегулярна мережа побудована з них і два ізохрони побудовані по сітці інтерпольованих значень.

Однозначним плюсом цього методу - є можливість працювати практично з будь-яким провайдером. Також маючи сітку точок з інтерпольованими часовими відстанями, можна малювати скільки завгодно ізохронів, без додаткових звернень до сервера за даними. Хоч описаний метод і може дати досить прийнятні результати для певних задач, але він має суттєві недоліки. Так як опорні точки ставляться у випадкових місцях, то вони можуть потрапити у важкодоступні місця, десь посеред лісу наприклад. Це може сильно спотворити результат. Тому потрібно робити якісь евристичні алгоритми, що будуть їх відсіювати. Можна наприклад використати середньоквадратичне відхилення для кожного кола і відсіювати всі точки, які виходять за межі відхилення. Проте це евристичний алгоритм, і можуть бути ситуації, коли при такому підході відсіюється частина

нормальних точок. При збільшенні точності необхідно збільшувати кількість точок, а відповідно кількість звернень до API, тому витрати можуть рости в геометричній прогресії. При цьому також, може суттєво збільшитись час, необхідний для генерації ізохрона, особливо якщо генерація ізохрона відбувається у браузері. Також точність ізохорну можна покращити, якщо збільшити кількість точок у сітці точок з інерпольованими значеннями часової відстані, але це також може суттєво вплинути на час генерації.

1.5.2 Спеціалізовані алгоритми

У спеціалізованих алгоритмах використовується добре відомий алгоритм Дейкстри для пошуку найкоротшого шляху у графі. Проте є одна суттєва проблема - швидкодія. Складність алгоритму Дейкстри складає O(n2). Де n - це кількість вершин у графі. На великих відстанях граф може складатись з десятків мільйонів вузлів. Тому є багато модифікацій і покращень для різних задач але всі вони базуються на алгоритмі Дейкстри. Наприклад можна зробити деякі попередні прорахунки, щоб у момент запиту, максимально швидко згенерувати ізохрон, проте це буде вимагати більше оперативної пам’яті та дискового простору. Різні двигуни маршрутизації(routing engines) по різному підходять до цієї задачі.

Один із поширених підходів називають багаторівневий Дейкстра(Multi-Level Dijkstra). В літературі він також зустрічається як багаторівневі корзини(Multi-Level bucket). Його незалежно запропонували одразу троє вчених Dial, Wagner, and Dinitz [12].

Інший відомий підхід - Ієрархії скорочень(Contraction hierarchies) [6]. Дуже спрощено суть його заключається у тому, щоб зробити попередні прорахунки для найбільш відвідуваних проміжків графа (магістралі і т. п.). Ідея заключається у тому, що дороги ієрархічні, є великі магістралі, по яким найшвидше рухатись, потім менші дороги, і тупики, в які рідко хто виїжджає. Тримати інформацію про всі можливі маршрути для всіх вузлів графа вимагало б занадто багато дискового простору, але якщо розрахувати таку інформацію для ключових доріг, то це може

значно пришвидшити процес побудови маршруту для користувача. Ієрархії скорочень дають набагато кращий результат за багаторівневі корзини, якщо потрібно наприклад побудувати маршрут між двома країнами. Проте є і недоліки - ієрархія скорочень стає не така ефективна, якщо потрібно оновлювати інформацію в реальному часі, або близькому до реального часу. Наприклад, коли створилась пробка і т. П. Тому що потрібно знову перед прораховувати все.

1.5.3 Алгоритм Valhalla

Valhalla - один з популярних двигунів маршрутизації, з відкритим кодом. Як джерело даних переважно використовується Open Street Map. В скандинавській міфології Valhalla - це рай для звитяжних воїнів. До вибору такої цікавої назви підштовхнуло співпадіння акронімів ключових модулів двигуна зі скандинавською міфологією. Так наприклад ядро двигуна маршрутизації називається THOR (Tiled, Hierarchical Open Routing). Генерація інформації про подорож називається ODIN (Open Directions and Improved Narrative). А сервісний компонент називається TYR (Take Your Route).

Скандинавська міфологія - не єдине чим відома Valhalla. Цей двигун має значну підтримку від таких серйозних компаній як Mapbox і Tesla. Після закриття Mapzen, Valhala втратила фінансування, і Mapbox постарався найняти до себе більшість спеціалістів. Крім того Mapbox припинив підтримувати OSRM, ще один популярний двигун маршрутизації з відкритим кодом. А це означає, що один з найбільших провайдерів на ринку мап, уже переключився, або має плани використовувати Valhala. Крім того Valhala використовується в авто Tesla. А Tesla - одна з найдорожчих компаній у всьому світі. Це говорить про високу якість Valhalla і про те що вона буде розвиватись і підтримуватись ще багато років.

Інженери Valhalla виклали в мережу деякі подробиці, як працює їх алгоритм побудови ізохорну. Він чимось схожий на наближений алгоритм побудови ізохрону, який ми розглядали, але на відміну від першого не використовує інтерполяцію, а тому більш точний.

Спочатку створюється сітка точок, навколо вихідної точки. Потім цим точкам будуть проставляти часові значення і по цим значенням буде малюватись ізохрон. Це та частина де обидва алгоритми схожі. Проте часові значення у сітку будуть записуватись не на основі інтерполяції, а на основі алгоритму Дейкстри. На кожній ітерації точки з сітки, які перетинаються з сегментом дороги маркуються часом за який можна дістатись від вихідної точки. Якщо цей час менший, ніж потрібно для нашого ізохрону, то ітерації продовжуються. Як тільки експансія зупиняється - процес маркування точок також зупиняється, так як немає необхідності шукати час для точок, які точно будуть далі ніж нам потрібно для побудови ізохрону. Далі потрібно власне намалювати контур. Для цього використовується алгоритм CONREC, Пола Брука, який він зробив ще в 1980-их роках. Алгоритм уже є реалізований на всіх популярних мовах програмування [1]. У даному алгоритмі використовуються вузли, сусіди яких знаходяться по різні боки ізохрону. Наприклад ми малюємо ізохрон, для 100 секунд від вихідної точки, в процесі оцінки згенерованої сітки, ми знайшли точку, відстань до якої 99 секунд, а до її сусіда 102 секунди. Між цими точками і буде проходити контур ізохрона. Причому всі контури мають бути замкнені інакше не утвориться полігон.

1.6 Двигуни маршрутизації з відкритим кодом

На ринку є багато платних провайдерів, як Google Maps, так і open source рішень, які можуть бути більш ефективними з точки зору витрат, на великих об’ємах, або давати кращий функціонал, більшу свободу. Наприклад, незважаючи на те, що Google Maps - найбільший провайдер мап на ринку, але в нього немає API для побудови ізохронів. Деякі платні провайдери використовують open source рішення, але беруть гроші, за те що надають зручний сервіс для користувача. Так працює наприклад Mapbox. На ринку є кілька відомих opensource рішень у кожного є свої переваги і недоліки [15].

1.6.1 OSRM (Open Source Routing Machine)

Мабуть найпопулярніший на ринку на сьогоднішній день, проте зараз втрачає популярність. Написаний на C++. Дуже ефективний з точки зору швидкодії. Має API для всіх популярних мов програмування. Активно підтримувався Mapbox, проте останні кілька років популярність падає, так як основні Mapbox був основним контрибутором, а зараз вони переключились на Valhalla. Тому активної розробки нових фіч не відбувається. Крім того, з коробки OSRM не вміє будувати ізохрони. Є бібліотека, яка додає цей функціонал, але вона в експериментальному режимі і останній коміт був у 2016-у році. Доречі бібліотека знаходиться у відкритих репозиторіях Mapbox (https://github.com/mapbox/osrm-isochrone). Схоже, що Mapbox не дочекався повноцінної підтримки ізохронів у OSRM і переключився на Valhalla, так що навряд чи варто її очікувати у найближчий час. Крім того в OSRM немає підтримки публічного транспорту. Незважаючи на вказані недоліки OSRM зрілий і надійний продукт, який використовується розробниками вже понад 10 років.

1.6.2 Graphhopper

Хоч і являється open-source рішенням, але дуже вузькоспеціалізованим і зрештою для комерційного використання, а не для широкої аудиторії. Graphhopper зосереджений на вирішенні дуже складних проблем маршрутизації авто. Використовується наприклад для планування маршрутів комерційних авто. Має хорошу швидкодію. Вміє будувати ізохрони. За свою продуктивність платиться надмірним використанням ресурсів, особливо оперативної пам’яті.

1.6.3 Valhalla Routing Engine

Найперспективніша на думку автора система на сьогоднішній день, так як має підтримку Mapbox і Tesla. Вміє будувати ізохрони з коробки, гнучка, має широкий функціонал, легко інтегрується з python. З точки зору швидкодії повільніша за OSRM, проте для більшості випадків це не критично. Також до недоліків можна віднести важкість налаштування для новачків, через відсутність

обширної документації. Проте є готові збірки в докері, які можуть значно полегшити початкову інсталяцію і зменшити поріг входу в технологію.

1.6.4 Openrouteservice

Один з найстаріших двигунів на ринку. Свій шлях почав у Німеччині, в 2008 році, університет Heidelberg. Початкова реалізація була в PostgreSQL, проте потім став розширеною версією Graphhopper. Він дає більше свободи ніж Graphhopper, наприклад, ви можете задати, щоб маршрутизатор уникав магістралей, або платних доріг. За це платить швидкодією. Тому хоч і базується на Graphhopper, але працює трохи повільніше. Проте це все одно одне з найшвидших рішень на ринку. Так само як і Graphhopper має велике споживання ресурсів. Наприклад якщо ви хочете будувати маршрути по всьому світу, то вам потрібно як мінімум 128 ГБ оперативної пам’яті.

1.6.5 PgRouting

PgRouting - це розширення до найпотужнішої бази даних з відкритим кодом PostgreSQL. PostgreSQL - це швейцарський ніж у світі баз даних. Дуже широкий функціонал, дуже просто, проте спеціалізовані інструменти дають кращий результат. Але якщо для вашого продукту достатньо pgRouting, то з точки зору інтеграції - це найпростіше рішення. Встановлюється pgRouting однією командою - CREATE EXTENSION pgRouting. pgRouting працює в парі з PostGIS, розширення для PostgreSQL, яке дає підтримку роботи з географічними об’єктами. Звичайно швидкодія у pgRouting найгірша з усіх, проте якщо вам не треба будувати маршрут через всю країну за долі секунди, а таких маршрутів насправді не так багато будується пересічними користувачами, то це може бути хороше рішення як мінімум для старту. Також pgRouting підтримує побудову ізохронів.

1.7 Рекомендаційні системи

Так як ми маємо на меті побудувати рекомендаційну систему, то потрібно розібратись з основними поняттями.

Рекомендаційна система - це підклас систем фільтрації інформації. Метою рекомендаційної системи є фільтрація з усього переліку об’єктів, тих об’єктів, яким користувач надасть перевагу з більшою вірогідністю. Часто ці об’єкти мають рейтинг і можуть бути впорядковані відповідно до рейтингу. Всепроникність варіативність і актуальність рейтингових систем важко переоцінити. Вони оточують нас всюди. Від банальної фільтрації пісень за автором, до колаборативних оцінок фільмів на IMDB та machine learning на Youtube. В 2009 році Netflix зробив конкурс з призом 1 000 000 $ для команди, яка зробить принаймні на 10% кращий алгоритм рекомендації фільмів ніж у них.

Чим більше рекомендаційна система знає про вас - тим кращий результат може дати на виході. Раніше виділяли явні і неявні методи збору даних. Але зараз мабуть формулювання неявного збору даних трохи застаріле, так як все більше країн приймають закони на кшталт GDPR, в тому числі і Україна. Згідно GDPR(General Data Protection Regulation) будь-який неявний для користувача збір даних являється протизаконним. Фактично є більш очевидні для пересічного користувача методи збору даних і менш очевидні. Наприклад, користувач погодився при заході на сайт, що його поведінка буде записуватись, з метою покращення рекомендацій для нього, але може не знати яка саме поведінка і як це впливає. І по великому рахунку більшості користувачів такі деталі не цікаві. Щодо очевидних методів збору даних - це може бути наприклад пряме запитання, яке авто ви використовуєте?

Типові підходи в рекомендаційних системах:

- Групова(колаборативна) фільтрація. Метод основується на гіпотезі, що люди яким були цікаві одні й ті самі речі в минулому, скоріш за все будуть цікаві ті самі речі і в майбутньому. Простими словами - спільні інтереси.

- Фільтрація об’єктів за певними признаками. Наприклад, якщо ви вказали, що вас цікавить житло у Вінниці, то система може відфільтрувати об’єкти з інших міст.
- Мультифакторна рекомендаційна система. Оцінка за багатьма критеріями. Наприклад, ви можете вказати у рекомендаційну систему, що для вас важлива площа квартири, а також важлива відстань, до найближчої станції метро і стан ремонту. Рекомендаційна система може на основі цих факторів, через ваги, порахувати рейтинг кожної квартири і показати їх у відповідності до цього рейтингу.
- Оцінка ризиків. Більшість рекомендаційних систем не оцінюють ризики рекомендації невдалого об’єкта. Наприклад ви шукаєте авто, і ви мабуть хочете купити його дешевше, але якщо якесь авто надто дешеве, то рекомендаційна система може його оцінити як ризикове і понизити у рейтингу, щоб покупець на емоціях не купив кота в мішку.
- Мобільні рекомендаційні системи. Використовують дані з смартфонів для рекомендацій. Наприклад дані GPS.
- Гібридні рекомендаційні системи. Більшість рекомендаційних систем являються гібридними, так як задачі з кожним днем стають все складнішими і однієї методики часто буває недостатньо щоб задовольнити всі вимоги.
- Сесійні рекомендаційні системи. Системи які базуються на сесійних даних користувача.
- Навчання з підкріпленням та інші методи машинного навчання.

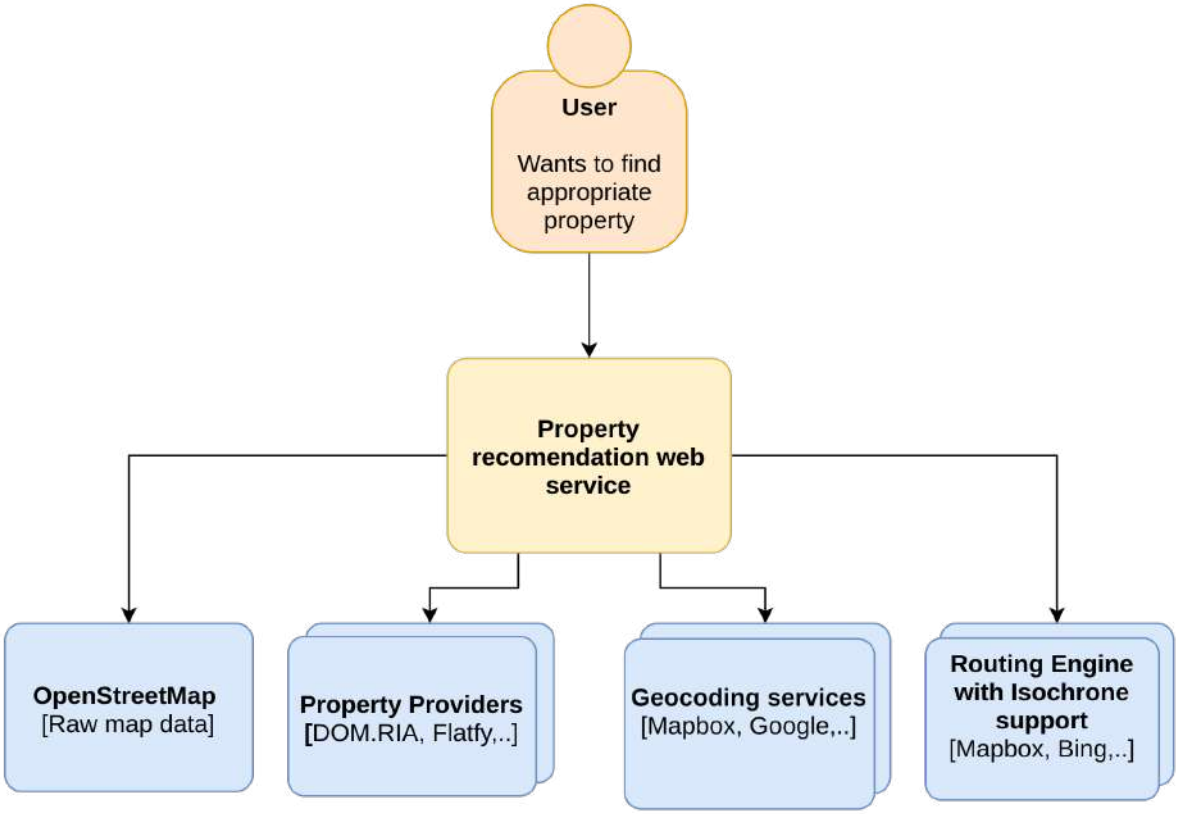
Звичайно це не повний перелік, так як рекомендаційні системи розвиваються дуже стрімко. Але чи можуть ізохрони виступати у ролі рекомендаційних систем? Так як ізохрон візуально відокремлює одні об’єкти на мапі від інших, то це можна назвати фільтрацією. Побудова ізохронів відбувається на основі вподобань користувача - його точок інтересу. Якщо користувач введе декілька точок, то це вже буде мультифакторна рекомендаційна система. Якщо до прототипу додати ще

можливість наприклад фільтрувати об’єкти по кількості кімнат, то це вже буде гібридна рекомендаційна система. Також маючи матрицю точок доступності можна легко відсортувати всі об’єкти в рамках ізохрону в залежності від часу і це вже буде рекомендаційна система з ранжуванням. Враховуючи всі вищенаведені фактори одназано можна сказати, що на ізохронах можна побудувати рекомендаційну систему.

Розділ 2. Архітектура

Ми розглянули основні технології специфічні для нашої доменної області, а саме: клієнтські бібліотеки для виводу мап у браузері, методи виводу великої кількості об’єктів на мапі без деградації продуктивності, геокодування, автодоповнення, алгоритми побудови ізохронів, двигуни маршрутизації. Також ми розглянули потенційні джерела даних для нашої системи. В цьому розділі представлена архітектура системи в цілому, з поясненням вибору компонентів та архітектурних патернів. Буде представлена контекстна діаграма, загальна компонента діаграма, кілька більш детальних компонентних діаграм. В цілому архітектура системи cloud-agnostic, тобто немає жорстких залежностей від якогось конкретного провайдера хмарних послуг і може бути реалізована на будь-якому публічному чи приватному хмарному постачальнику. Більше того, система може бути повністю реалізована на базі opensource рішень. Деякі деталізовані діаграми будуть з прив’язкою до AWS(Amazon Web Services), як приклад, для більш наглядної демонстрації, як дана архітектура може бути реалізована у конкретного провайдера, але ці діаграми можуть бути адаптовані і під інших провайдерів хмарних послуг. Після розгляду архітектури, ми реалізуємо і розглянемо прототип. Прототип буде реалізований на спрощеній архітектурі. Основна задача прототипу - перевірити досяжність реалізації більш складної архітектури та надати потенційним користувачам можливість оцінити корисність рекомендаційної системи пошуку житла на мапі з використанням ізохронів.

2.1 Контекстна діаграма



Діаграма 2.1.1 - Контекстна діаграма рекомендаційної системи пошуку підходящого житла.

На контекстній діаграмі зображено основні зовнішні сервіси, які взаємодіють з рекомендаційної системою пошуку житла, а також користувач. Стрілки на діаграмі зображають потік керування. Потік даних відбувається відповідно у зворотному напрямку.

Основна мета користувача(Job to be Done) - знайти собі підходяще житло для оренди чи покупки. Допомогти йому у цьому - задача нашої рекомендаційної системи. Для цього нашій системі потрібно взаємодіяти з іншими сервісами та провайдерами даних.

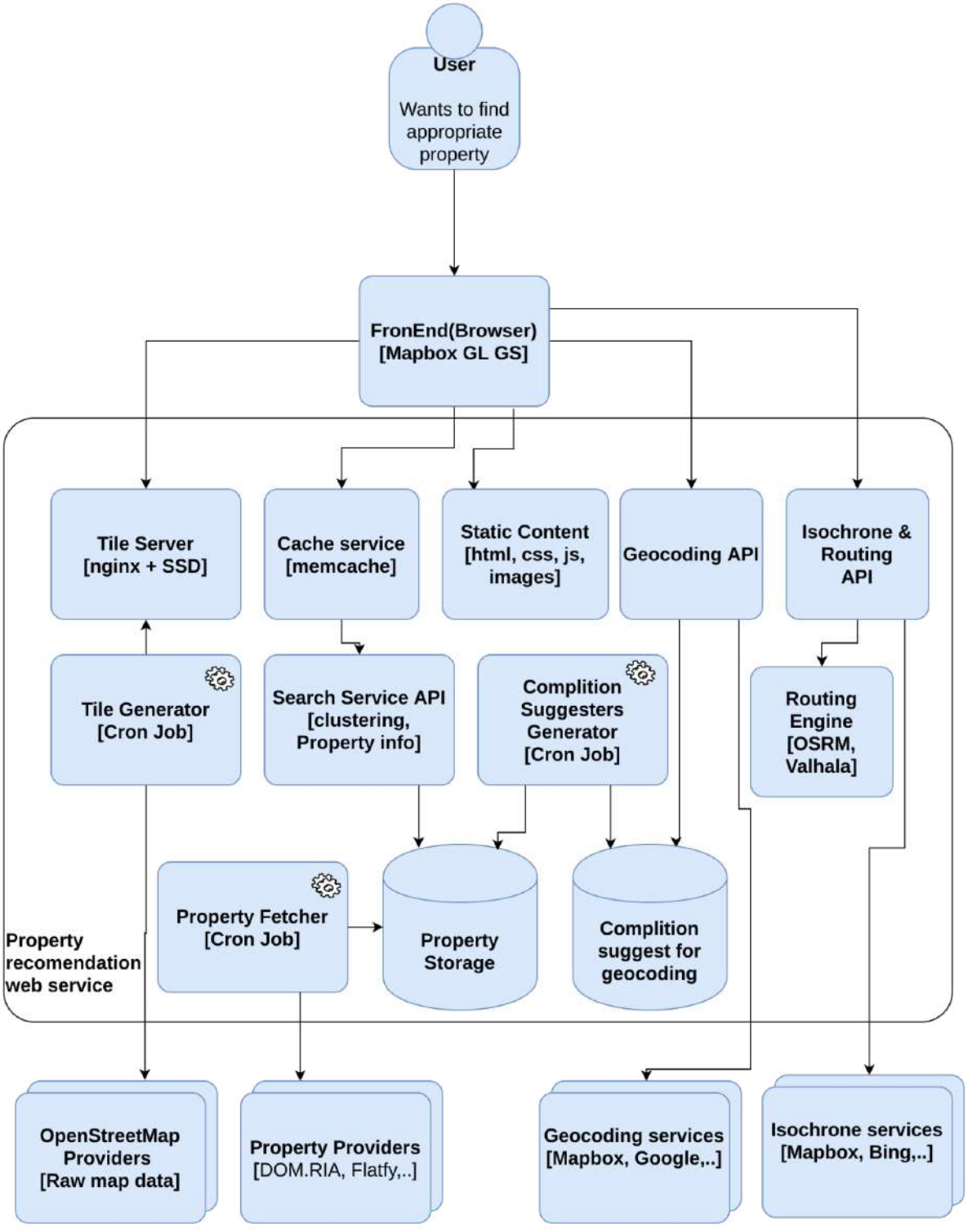
Головний потік даних - це дані про нерухомості, без цих даних наша система не зможе функціонувати. Ми вже розглядали, детально у розділі 1.1 доступні в Україні провайдери даних про житло і способи отримання цих даних.

Також нам необхідні сирі дані з OpenStreetMap. OpenStreetMap(OSM) - це відкритий проєкт, який займається збором, збереженням та розповсюдженням загальнодоступних геопросторових даних. Є кілька сервісів, які надають можливість безкоштовно завантажити актуальні дані OpenStreetMap: Geofabrik, Overpass API, Planet OSM,.. та інші. Ці дані можуть бути використані для багатьох цілей: створення шару “підложки” для мапи, для сервісу пошуку точок інтересу, геокодування, створення ізохронів, маршрутів… Ми також можемо використати готові сервіси для деяких з цих задач, або комбінувати дані з різних сервісів, проте дані з OSM дають можливість зробити повністю opensource архітектурне рішення.

Також нам потрібен провайдер для геокодування, для того щоб конвертувати точки інтересу користувача у довготу і широту, і відповідно мати можливість відмітити їх на мапі. Ми можемо створити власний сервіс геокодування, або використати готові рішення, наприклад від Google Maps. Google Maps має найбільш повну базу гео об'єктів, тому може дати кращі результати, але варто також брати до уваги вартість сервісу - 2.83 $ за тисячу запитів.

Останнім, важливим сервісом для нашої системи являється двигун маршрутизації. Тут, так само, як і в ситуації з геокодуванням, може бути використане openxource рішення, або уже готовий сервіс, як наприклад Mapbox, Bing, Google Maps. Google Maps хоч і не вміє малювати точні ізохрони, але дає більш точну інформацію по конкретному маршруту, так як має найбільш повну інформацію про трафік.

2.2 Компонентна діаграма



Діаграма 2.2.1 - Компонентна діаграма рекомендаційної системи пошуку підходящого житла.

Розглянемо структуру нашого веб сервісу більш детально на компонентній діаграмі. Основні патерни, які використовувались при проектуванні - це фасад і агрегатор.

Патерн фасад приховує нижні шари архітектури і дає консистентний API незалежно від них. Це дає нам можливість легко змінювати архітектуру внутрішніх сервісів, чи змінювати зовнішні провайдери даних, без внесення змін на фронтенді. Розглянемо для прикладу сервіс, який створює ізохрони. Як універсальний API він може повертати полігон у форматі GeoJson. Ми можемо зробити MVP версію нашого сервісу з використанням ізохронів, які будує Mapbox, а потім з часом переключитись на власний пошуковий двигун, щоб оптимізувати витрати. Або використовувати для ізохронів власний пошуковий двигун, а для побудови маршруту Google Maps, так як він має більш повну інформацію про трафік і дає більш точну інформацію. Але використовувати Google Maps для ізохронів мабуть не найкраще рішення, так як у них немає такого API, а розрахунок ізохронів за допомогою Distance Matrix API дає неточні результати із-за інтерполяції, а для покращення точності потрібно робити більше запитів на Distance Matrix API, що дуже сильно впливає на вартість платформи в цілому. Також фасад знімає жорстку залежність від одного провайдера і робить платформу більш стабільною. На базі фасаду можна реалізувати логіку, коли один сервіс провайдер по якійсь причині не доступний - використовувати інший. Отже фасад полегшує легкість внесення змін у майбутню архітектуру для покращення якості системи, оптимізації витрат, покращення стабільності роботи системи. Якщо говорити про більш конкретні деталі, то практично всі API фасади можна реалізувати за допомогою лямбда функцій на AWS. Лямбда функції оплачуються лише той час, коли вони працюють, а їх реалізація набагато простіша ніж створення віртуальної машини чи докер контейнера.

Паттерн агрегатор допомагає нам комбінувати і доповнювати дані. Розглянемо патерн на API геокодування. У нашій системі може бути кілька джерел даних для точок інтересів, наприклад ми можемо брати результати з OSM даних,

доповнювати їх якимись своїми локальними даними, даними від провайдерів нерухомості(наприклад назва ЖК), а якщо ми не можемо знайти що вводить користувач, то звернутись до Google Maps і доповнити відповідь ще їх даними. Отже паттерн агрегатор - агрегує дані з різних джерел. Це дає нам можливість зробити якісний пошук географічних об’єктів, щоб задовольнити потреби користувача.

Користувач взаємодіє з рекомендаційною системою через браузер. В браузер, з серверів, які займаються статичним контентом завантажується необхідні ресурси: html, css, images,.. і в тому числі наша основна клієнтська бібліотека - Mapbox GL GS. Для цього краще використовувати спеціалізовані сервіси доставки статичного контенту CDN(Content Delivery Network), або спеціалізовані веб сервери для роздачі статичного контенту, які здатні тримати великі навантаження, nginx наприклад.

Після завантаження Mapbox GL GS має підгрузитись основна підложка мапи з усіма об’єктами, крім наших об’єктів нерухомості. Генерацією плиток для мапи буде займатись окремий сервіс. Він буде завантажувати сирі дані з OSM, генерувати з них плитки та складати їх на сервіс, який буде займатись їх роздачею. Знову ж таки це може бути або CDN, або nginx з ssd диском для того щоб тримати великі навантаження, так як завантаження мапи навіть для одного користувача потребує завантаження десятків плиток, а також вони завантажуються при прокрутці мапи, збільшенні або зменшенні.

Для генерації плиток мапи, можна використати популярний opensource продукт - Mapnik. Він вміє генерувати з OSM даних генерувати растрові зображення(плитки мапи). Причому ці зображення можна стилізувати і зобразити впізнаваний сервіс з унікальним дизайном. За допомогою Mapnik можна також генерувати векторні плитки, якщо встановити додатковий модуль. Але це не є його основною областю використання, так як Mapnik по суті являється модулем, який рендерить зображення, а векторні плитки - це не зображення у класичному розумінні, це опис векторів з додатковими даними, у компактному бінарному

форматі протобаф. Mapnik має хорошу інтеграцію з іншими opensource модулями. А також підтримує такі популярні мови програмування як Nodejs та Python.

Також нам потрібні об’єкти нерухомості на мапі. За їх доставку, фільтрацію, кластеризацію та надання додаткової інформації буде відповідати Search Service. Він може передавати кластеризовані результати пошуку у вигляді GeoJson або більш компактного протобаф, в залежності від об’єму даних. Для серверної кластеризації можна використати бібліотеку supercluster. Дані будемо брати з бази Property Storage. База Property Storage зберігає дані про об’єкти нерухомості. Потрапляють вони туди, за рахунок сервісу Property Fetcher. Сервіс Property Fetcher бере дані з відкритих API від різних провайдерів. Так як дані можуть мати різну структуру, то наша БД має мати можливість зберігати не чітко структуровані дані, а також бажана підтримка геозапитів, тобто фільтрація по заданим координатам. Для таких цілей добре підходить Elastic Search, він з коробки підтримує геозапити, добре масштабується і підтримує чітку і нечітку схеми збереження даних. Також можна використати PostgreSql, з додаванням необхідних модулей. Геозапити потрібні для того, щоб витягувати дані лише для тих плиток, які відображаються у користувача.

Так як операція кластеризації досить затратна по ресурсу, то є сенс поставити кешуючий сервіс, перед сервісом пошуку. Це знизить навантаження на базу даних, використання серверних ресурсів, а також час на генерацію відповіді. Важливий момент для кешів заключається у тому, що запити мають генеруватись по краям плиток, а не по краям видимої мапи у користувача, інакше здвиг мапи навіть на один піксель буде вимагати генерації нових кластерів. Як наслідок кеші будуть роздуватись, відсоток потрапляння у кеш буде дуже малий, система буде працювати повільніше ніж з кешами. А вартість системи буде дуже висока, так як для кешування зазвичай використовують оперативну пам’ять, а це найдорожча пам’ять. Також можна генерувати кеш не по краям плитки, а для всієї мапи на кожен рівень збільшення мапи. І потім відсікати з усіх даних, ті які входять у видиму користувачу область. Для кешування можна використовувати memcache

або redis. У випадку генерування кешів на всю мапу, можна використовувати більш складну базу, з підтримкою геоіндексів, для ефективного відсікання. Наприклад PostgreSQL з модулем PostGIS, який додає підтримку GeoJson і зручних функцій для роботи з ним.

Також у нашій системі присутній сервіс геокодування з функціональністю автодоповнення. Окремий сервіс зроблений для того, щоб не прив'язуватись до специфіки API провайдерів геокодування. Таким чином ми можемо комбінувати результати від різних провайдерів, змінювати провайдерів в залежності від економічної доцільності, або взагалі відмовитись від них і використовувати лише свій власний сервіс на основі open source модулів. Власний сервіс нам потрібен також для збагачення даних зі сторонніх сервісів, своїми власними даними. Цей сервіс може бути реалізований, наприклад, за допомогою Elasticsearch completion suggester. Elasticsearch дає можливість пошуку підказок з урахуванням контексту, а також геолокації, крім того, може виправляти помилки користувача при введенні на задану відстань левенштейна. Відстань левенштейна - це мінімальна кількість правок, щоб зробити з неправильної фрази у правильну. За рахунок того, що база підказок зберігається в оперативній пам'яті - гарантує високу швидкодію. Є також готові рішення від cloud провайдерів.

Остання група компонентів відповідає за генерацію ізохронів і маршрутів. Цим може займатись як один сервіс, так і декілька. Більш детально ця частина описана як приклад патерну фасад. Технології і методи генерування ізохронів описані детально у розділах 1.5 Алгоритми побудови ізохронів та 1.6 Двигуни маршрутизації.

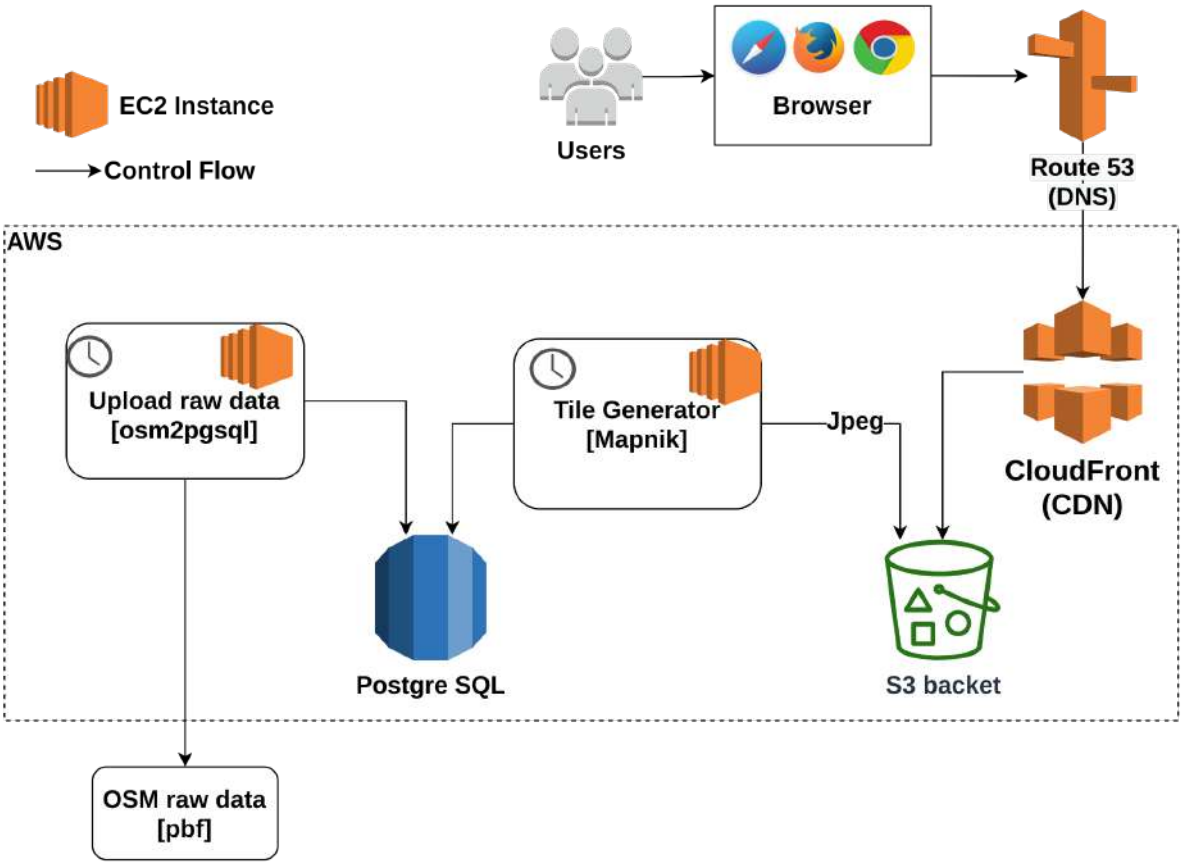
2.3 Мова програмування

Так як ми створюємо веб сервіс, то на клієнтській частині(браузері) ми можемо використовувати тільки Javascript. Можливі поправки на typescript або якісь інші мови, але потім з них все одно буде згенерований Javascript. Для серверної частини логічним продовженням стане nodejs - серверна реалізація

Javascript. Nodejs добре себе зарекомендував для побудови інтеграційних API(фасадів). За рахунок асинхронності він дає хорошу швидкодію. За рахунок однопоточності - простоту написання коду. А за рахунок того, що це всім добре відомий javascript - популярність, і в нашому випадку можливість задіяти fullstack розробників. За рахунок своєї популярності nodejs має дуже хорошу інтеграцію з більшістю популярних бібліотек, модулів та сервісів. У нашому випадку це можуть бути лямбди на AWS, OSRM, ELasticSearch, Superclasterer…

2.3.1 Компонента діаграма генератора плиток мапи на AWS

Розглянемо більше детально архітектуру генератора плиток для мапи на прикладі AWS. Дані компоненти можна також адаптувати під інших провайдерів хмарних послуг.



Діаграма 2.3.1. Генератор плиток для мапи

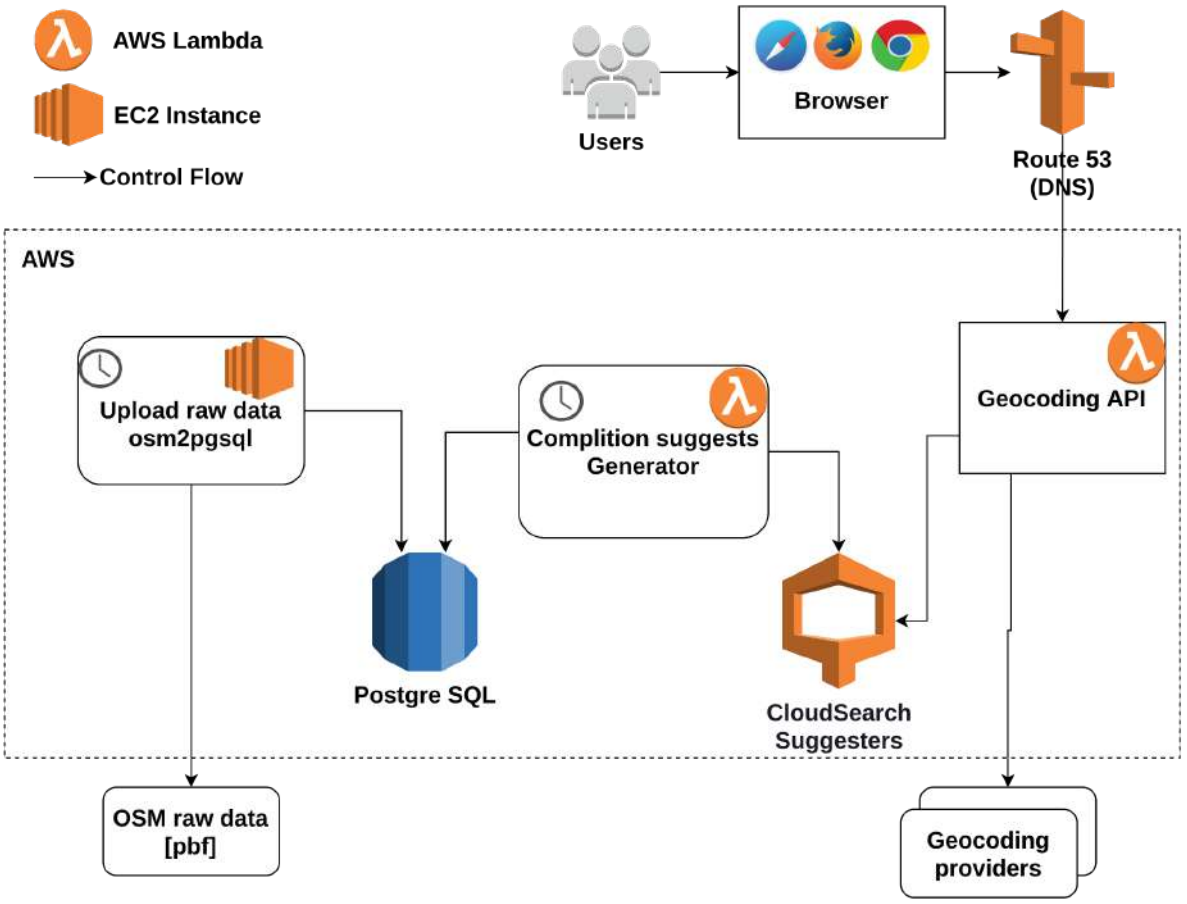
Перший модуль osm2pgsql завантажує по розкладу OSM дані у форматі pbf і записує їх в базу даних PostgreSQL. AWS підтримує дану БД, як SaaS рішення. Дані можна завантажувати і в інших форматах, але так як об'єм даних дуже великий, то протобаф буде найбільш доцільним форматом. Для України розмір OSM даних складає порядку 600 МБ у форматі протобаф. Завантажити дані можна з Geofabric або інших сервісів. На AWS цей модуль можна виконати на EC2(Elastic Container).

На наступному кроці Mapnik, модуль який вміє з geojson генерувати растрові зображення, власне і генерує плитки. Він їх генерує також за розкладом і може бути виконаний як EC2. Плюс в тому, що даний модуль, так само як і osm2pgsql може бути повністю зупинений в час коли не працює, а тому ми будемо оплачувати лише той час, який ми використовуємо даний сервіс для генерації плиток. Плитки складаються у сховище S3 bucket. Воно вміє зберігати зображення та інші дані. S3 bucket підключається до Amazon CloudFront. Amazon CloudFront - це CDN на базі AWS, яка буде займатись доставкою зображень до кінцевого користувача.

Останній компонент - Route 53. Він опційний. Це DNS(Domain Name Server) за допомогою якого можна змінити назву домену з стандартної, згенерованої CloudFront, на свій домен. Для прикладу url http://d111111abcdef8.cloudfront.net/tile_123.jpg можна змінити на http://example.com/tile_123.jpg.

2.3.2 Сервіс геокодування

Сервіс геокодування буде використовуватись для пошуку точок інтересу а також конвертації їх у широту та довготу для відображення на мапі і подальшої роботи з ними. Для прикладу користувач знає, що адреса місця роботи знаходиться за адресою Академіка Янгеля 4, але навряд чи він знає широту і довготу. Тому нам потрібен сервіс геокодування. В даній архітектурі використаний паттерн агрегатор, де результати геокодування з даних OSM за необхідності збагачуються даними від інших провайдерів.



Діаграма 2.3.2.1. Компонентна діаграма сервісу геокодування на базі AWS.

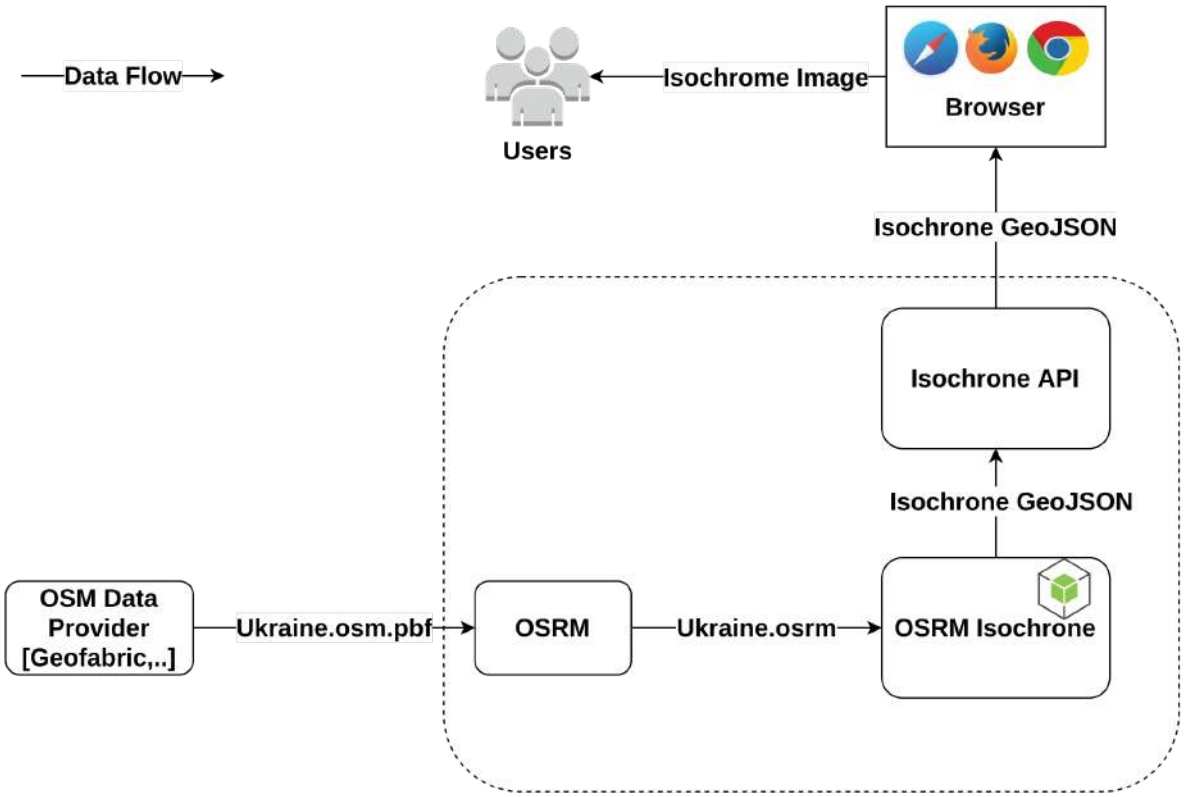
На даній діаграмі ми бачимо деякі вже знайомі компоненти з попередньої діаграми: osm2pgsql, PostgreSQL. Це не дублікати. Це ті самі компоненти, зображені на двох діаграмах для наглядності.

Крім знайомих компонентів також на діаграмі присутній компонент CloudSearch Suggesters. Це SaaS рішення для підказок автодоповнення. Нам він потрібен тому що наш сервіс геокодування має підтримувати автодоповнення. Дані в CloudSearch Suggesters будуть оновлюватись за розкладом лямбда функцією Completion Suggesters Generator. Лямбди на AWS можуть запускатись по розкладу, підтримують підключення до PostgreSQL на AWS і CloudSearch Suggesters.

Ще одна лямбда функція - Geocoding API. Ця функція компонує результати з CloudSearch Suggesters та інших провайдерів геокодування, за необхідності.

Дані користувачам віддаються через Route 53. На цій діаграмі це обов’язковий компонент, на відміну від попередньої, так як з браузера ми будемо отримувати дані через ajax запити, а для того, щоб вони працювали, потрібно, щоб доменне ім’я було однакове.

2.3.3 Сервіс генерації Ізохронів.



Діаграма 2.3.3.1. Компонентна діаграма сервісу генерації ізохронів.

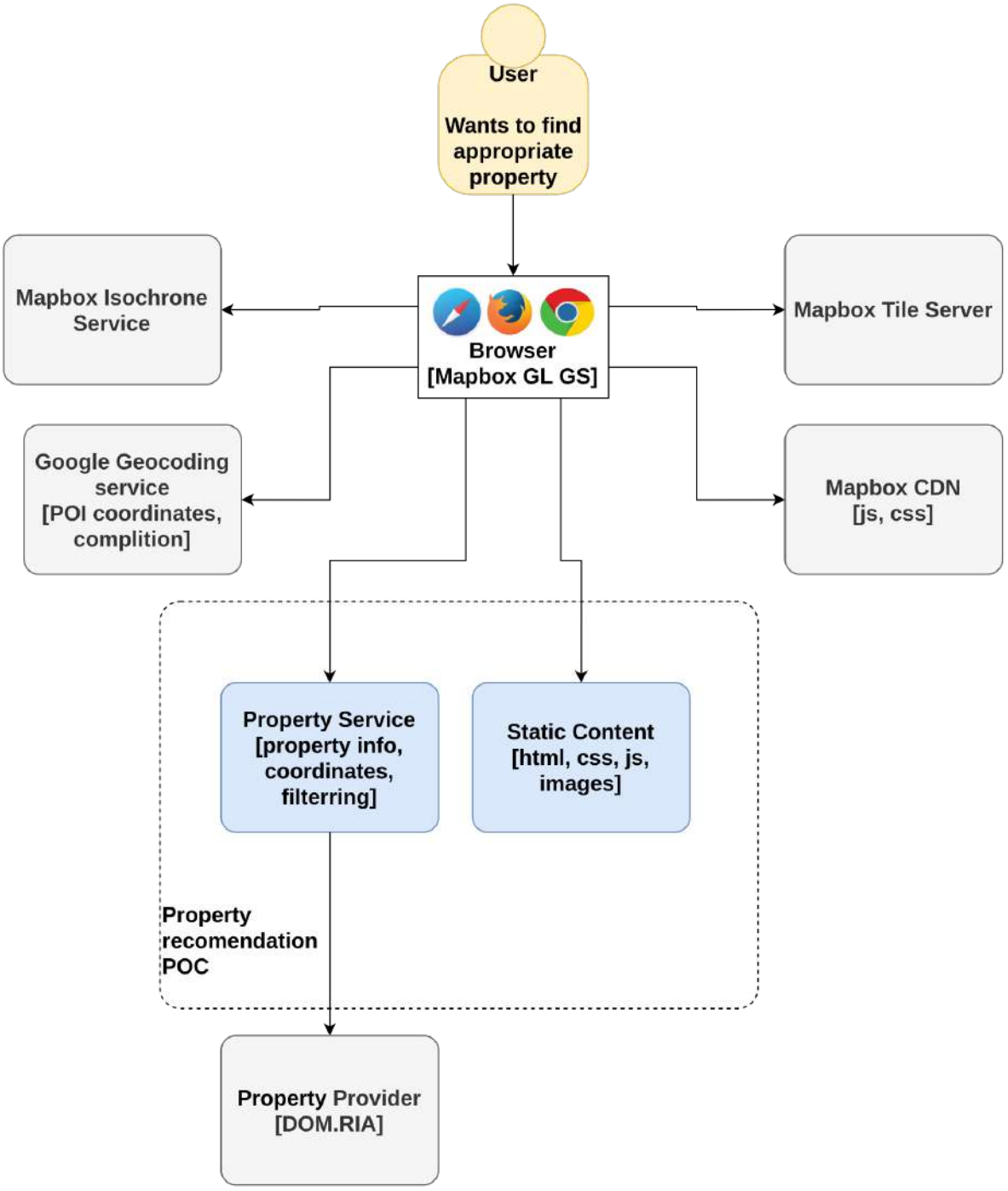
На даній діаграмі, на відміну від інших, стрілки зображують потік даних а не потік управління. На стрілках також вказано які дані передаються. На даній діаграмі, як приклад, використано рішення на базі OSRM(Open Source Routing Machine). Можливе використання також і інших двигунів маршрутизації з відкритим кодом, наприклад Valhalla.

Сирі OSM дані потрапляють на OSRM, звідки їх забирає модуль OSRM Isochrone написаний на nodejs, і генерує з них, за допомогою оптимізованого

алгоритму Дейкстри, ізохрон у вигляді полігона в форматі GeoJson. OSRM Isochrone підтримує два алгоритми оптимізації алгоритму Дейкстри: Multilevel Dijkstra та Contraction hierarchies. Детальніше ці алгоритми описані у розділі 1.1.1 Спеціалізовані алгоритми побудови ізохронів. Створений полігон передається на Isochrone API, яке являється фасадом. Потім браузер за допомогою клієнтської бібліотеки(Mapbox GL) малює, з GeoJson полігону, ізохрон на мапі. Це дає користувачу візуальне представлення досяжності його точок інтересу від об’єктів нерухомості.

Розділ 3. Прототип

3.1 Архітектура прототипу



Діаграма 3.1.1. Архітектура прототипу.

Архітектура прототипу максимально спрощена. Вона нестійка до відмов, не може витримувати велику кількість користувачів. Основна задача прототипу - надати потенційним користувачам можливість оцінити корисність рекомендаційної системи на основі ізохронів, а також оцінити доцільність розробки більш складної архітектури і перспективність технологій. Прототип підтримує лише desktop версію веб застосунку.

Підложка для мапи використовується з серверів Mapbox. Вона чудово працює, карта актуальна. Єдиний мінус - це висока ціна, у випадку, якщо запускати продукт не як прототип. Тому для комерційного запуску скоріш за все буде більш доцільно використовувати власний сервіс.

Так як підложка використовується векторна, то ми ще додатково завантажуємо стилі до мапи з CDN серверів Mabox, а також основну клієнтську бібліотеку для мапи - Mapbox GL GS. Знову ж таки, для комерційного продукту мабуть більш доцільно буде зберігати ці дані на власних серверах.

Для побудови ізохронів використаний сервіс Mapbox. В документації не розголошується інформація про алгоритм побудови ізохрону, але з огляду на те, що Mapbox активно розвиває проект з відкритим кодом Valhala, цілком логічно припустити, що вони використовуються той самий алгоритм. Більш детально цей алгоритм описаний у розділі 1.1.1. Як сервіс маршрутизації також використовується API Mapbox.

Для геокодування та автодоповнення використано сервіс Google Maps. Google Maps мають найбільш повну базу об’єктів, а тому дає кращі результати ніж аналогічний сервіс від Mapbox. Звичайно Google Maps дорожчий, але вони мають безкоштовний ліміт 200$ на будь-які сервіси на місяць. Цього більш ніж достатньо для прототипу.

Бекенд написаний на Node.js. Він складається з двох модулів: роздача статики і API для об’єктів нерухомості. Модуль роздачі статики надає html, js, css, а також картинки, необхідні для роботи веб застосунку. API об’єктів нерухомості надає координати всіх об’єктів нерухомості по заданим фільтрам, а також повну

інформацію про конкретний об’єкт нерухомості, для того, щоб виводити більш детальну інформацію по об’єкту, який зацікавив користувача. Так як фільтрація не є основною задачою прототипу, то реалізовані лише кілька базових фільтрів. У прототипі API підтримує наступні фільтри: область, тип операції(продаж, довгострокова оренда, подобова оренда), точність вказаних координат. До детальної інформації по об’єкту входить фото об’єкта, ціну, кількість кімнат, посилання на першоджерело інформації для більш детального ознайомлення. API виконує роль проксі. Дані не зберігаються.

Кластеризація відбувається у браузері. Для того, щоб забезпечити прийнятну швидкість кластеризації - об’єкти виводяться не по всій Україні, а по конкретній області. З тої ж причини максимальна кількість об’єктів лімітована до 10 тис.

3.2 Інтерфейси

Інтерфейс прототипу складається з двох частин: власне самої мапи з об’єктами нерухомості, а також панелі управління, з фільтрами і можливістю додавати точки інтересу. Ізохрони будуються навколо точок інтересу, це може бути наприклад місце роботи, навчання, тощо.

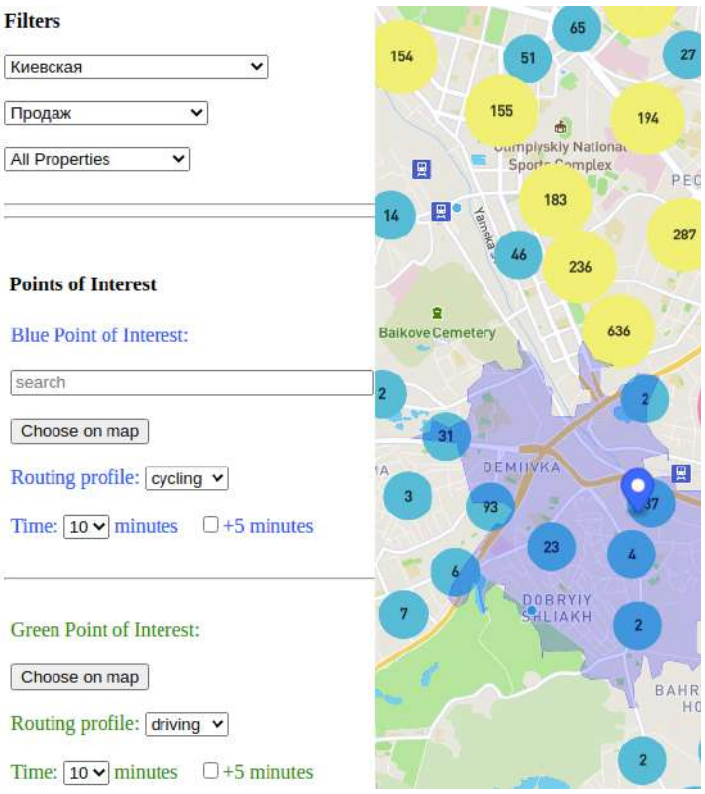


Рисунок 3.2.1. Інтерфейс прототипу. Панель управління зліва та мапа справа.

Для зручності та візуальної наглядності, розмір і колір кластера залежить від кількості об’єктів у кластері, а також в центрі кластеру відображається кількість об’єктів. Чим візуально більший кластер - тим більше у ньому згруповано об’єктів.

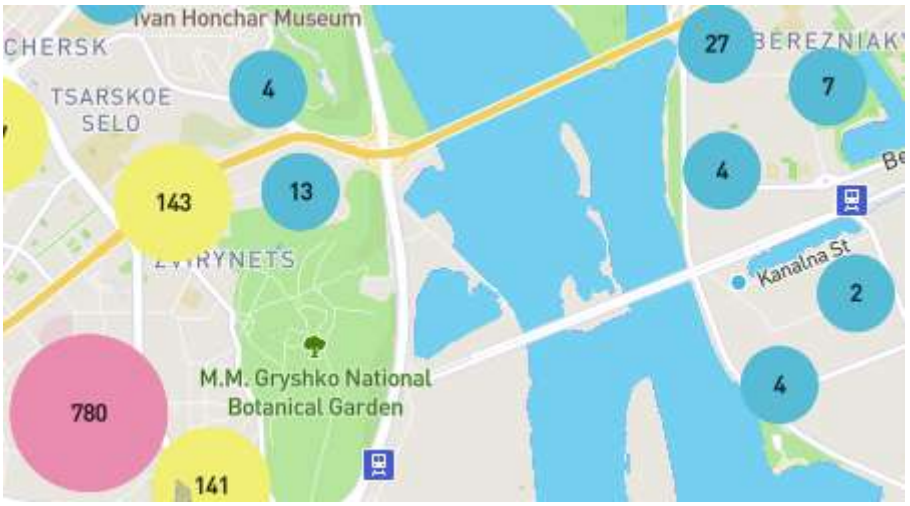


Рисунок 3.2.2. Кластери об’єктів нерухомості на мапі.

Точки інтересу можна задати двома способами. Перший - скористатись сервіс геокодування з автодоповненням. Тобто почати вводити назву бажаного об’єкту і вибрати з меню підказок. Так як не всі об’єкти можуть бути в базі Google Maps, а також іноді вони просто не так називаються, то є альтернативний спосіб. Вибрати точку на мапі, натиснувши відповідну кнопку і потім натиснувши на мишкою на мапі, в бажане місце.

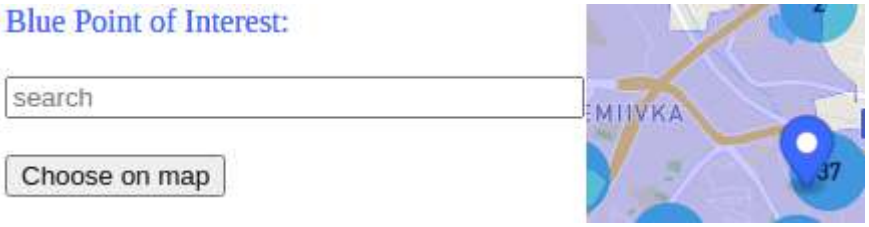


Рисунок 3.2.3. Способи вибору точок інтересу на мапі.

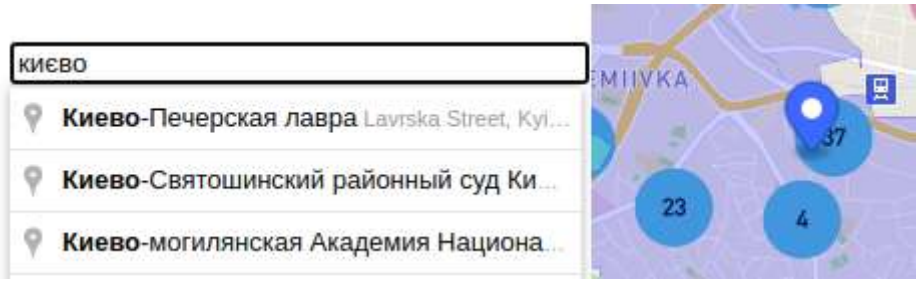


Рисунок 3.2.4. Вибір точки інтересу за допомогою сервісу автодоповнення.

В прототипі є можливість додати лише дві точки інтересу. Цього достатньо, щоб оцінити корисність ізохронів при пошуку житла, а також дає можливість шукати об’єкти, які задовольняють декільком критеріям через перетин ізохронів. Для зручності користування точки інтересу виділяються різними кольорами: зеленим і синім. Вони відмічені відповідним кольором на панелі управління. Також цим же кольором замальовується ізохрон, виділена сама точка на мапі, зафарбовується маршрут до вибраного об’єкту, і вказується додаткова інформація, яка стосується даної точки інтересу(час та спосіб пересування).

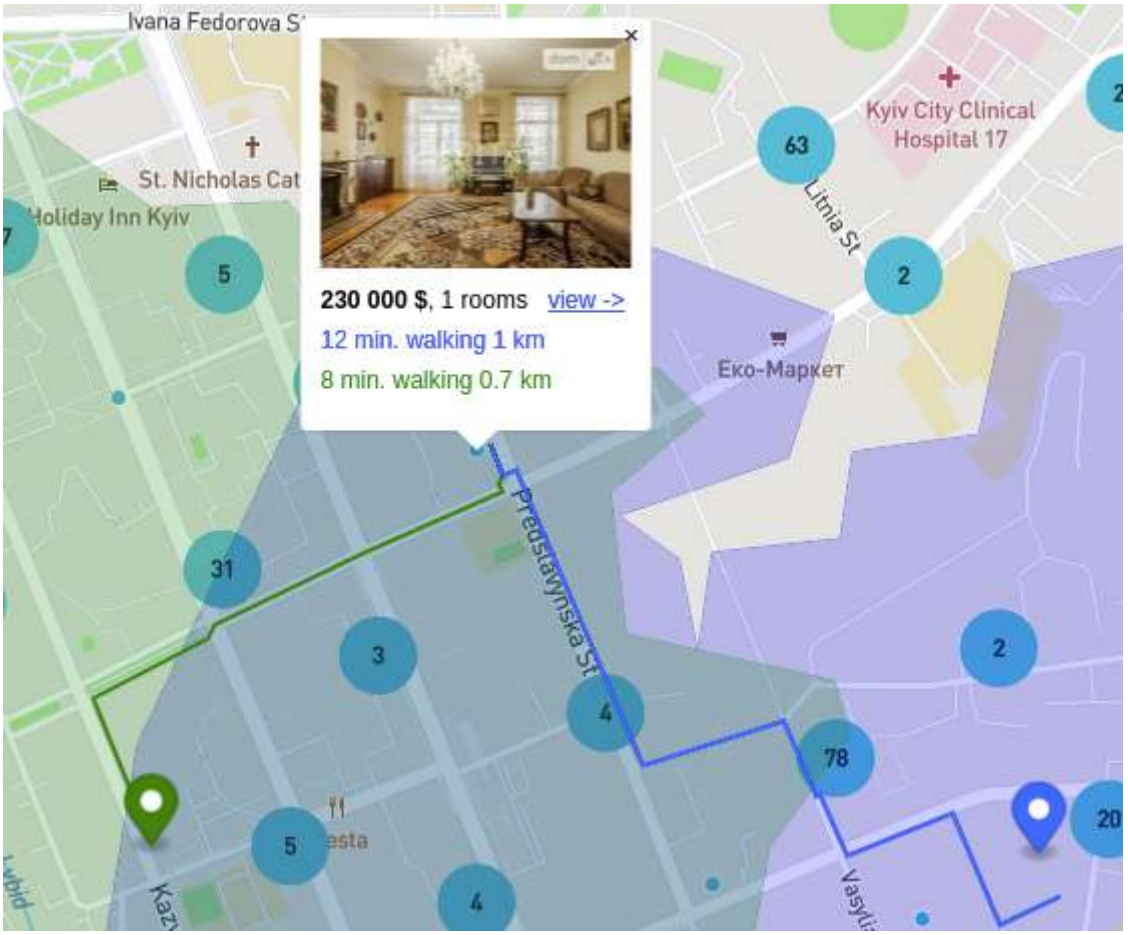


Рисунок 3.2.5. Дві точки інтересу(зелена та синя), перетин ізохронів, маршрут та додаткова інформація про об’єкт нерухомості.

При виборі точки інтересу - є можливість обрати один із трьох методів пересування: пішки, авто, велосипед. Також є можливість вибрати граничний час для ізохрону. Реалізовано за допомогою списку опцій, кратного 5 хв. Це зручний спосіб задати параметр, без необхідності щось друкувати на клавіатурі. Також є можливість намалювати додатковий контур в ізохроні. Це дає змогу наприклад відбирати об’єкти у певному часовому діапазоні, наприклад від 15 до 20 хв їзди. Або одразу розглядати об’єкти по двом критеріям. Наприклад спочатку я розгляну об’єкти, які знаходяться у внутрішньому ізохроні, але також буду розглядати об’єкти і в зовнішньому ізохроні, якщо там буде більш приваблива ціна. Для того, щоб відрізнити один контур від іншого, вони зображуються з різним рівнем прозорості.

Рисунок 3.2.6. Вибір методу пересування та часу.

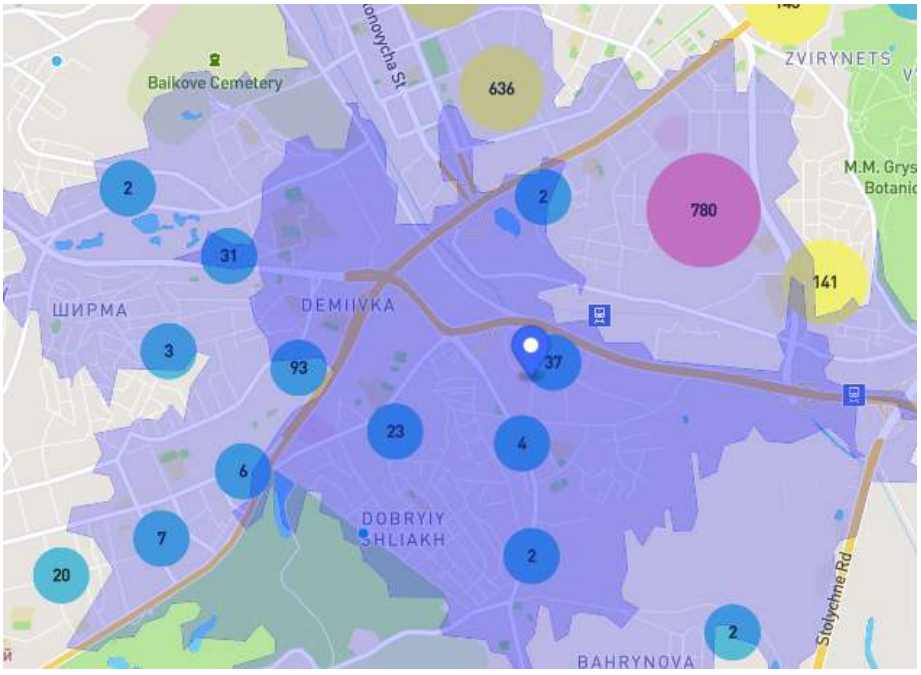


Рисунок 3.2.7. Подвійний ізохрон.

Після того як користувач задав основні параметри для рекомендаційної системи, він може ознайомитись з підходящими об’єктами на мапі. При натисненні на конкретний об’єкт нерухомості - відображається більш детальна інформація про нього: фото, ціна, кількість кімнат, а також посилання на першоджерело інформації, для того, щоб користувач міг ознайомитись з повною інформацією і зв’язатись з ріелтором чи власником житла. Також в інформері вибраного об’єкту нерухомості відображається час необхідний для того, щоб дістатись до всіх зазначених точок інтересу, вибраним методом пересування. Також відображаються маршрути по яким можна дістатись від обраного житла до всіх обраних точок інтересу. Це дає користувачу більш точну і повну інформацію для прийняття рішення щодо покупки чи оренди житла. Детально зображено на рисунку 3.2.5.

3.3 MVP

Звісно прототип не розрахований на повноцінний запуск. Розглянемо список допрацювань, який перетворить його у реальний продукт, яким зможуть повноцінно користуватись для пошуку житла.

Кількість точок інтересу має бути довільна, так як дві точки інтересу у багатьох випадках може бути недостатньо.

Мають бути всі базові фільтри, до яких звикло більшість користувачів на популярних сайтах нерухомості: ціна, метраж, тощо. Без цього функціоналу користувачі будуть вважати продукт неповноцінним, так як він не має базового функціоналу, який присутній на всіх сайтах нерухомості.

Брендинг і візуальна привабливість інтерфейсів також відноситься до базових речей без яких сервіс буде вважатись неповноцінним.

Адаптація інтерфейсу під мобільну версію. З кожним роком все більше користувачів використовують в основному мобільні пристрої для веб серфингу.

Для того щоб сервіс швидко працював на мобільних пристроях, потрібно реалізувати серверну кластеризацію. Це суттєво зменшить об’єм даних, що передаються і навантаження на процесор. Крім того це дозволить зняти примусове обмеження на максимальну кількість об’єктів, яка відображається на мапі.

Інформер об’єкту нерухомості має містити більше інформації про об’єкт: метраж, поверх, тощо. Для того щоб не змушувати користувача кожен раз переходити на сайт-першоджерело для ознайомлення.

Цих допрацювань буде достатньо, щоб рекомендаційна система виглядала як завершений продукт і отримала перших реальних користувачів. Звичайно це тільки початок і у випадку достатньої зацікавленості з боку аудиторії, необхідно буде суттєво змінювати архітектуру для того, щоб сервіс міг витримувати велике навантаження, розвиватись і бути економічно вигідним. Детальніше це питання розглянуто у розділі 2.2.

При розробці архітектури ми з’ясували, що за допомогою компонентного підходу до побудови архітектури, та можливості заміни провайдерів - можна відносно легко регулювати вартість кінцевої платформи. Яка вартість являється прийнятною - залежить від комерційної моделі і об’єму трафіка. Рішення може бути побудоване повністю на open source компонентах і місцевих, відносно недорогих хостинг провайдерах, або з використанням готових рішень від Google Maps та інших провайдерів на базі AWS. Відповідно буде змінюватись і ціна і час виходу на ринок готового продукта. В будь-якому випадку еластичність архітектури має покрити велику кількість патернів, а комерційну доцільність варто розглядати у кожному конкретному випадку окремо.

Висновки

Результати опитування

Після реалізації прототипу, було проведено анонімне опитування серед групи людей. Серед них були спеціалісти в сфері нерухомості, інформаційних технологій та цільова аудиторія рекомендаційної системи - люди, які зараз знаходяться в процесі пошуку житла, або мали такий досвід останні кілька років. Загальна кількість респондентів - 35. Звичайно це не можна вважати репрезентативною вибіркою, враховуючи масштаби потенційної аудиторії користувачів. Дане опитування не претендує на соціологічне дослідження - це швидше перевірка гіпотези. Тобто звичайно опитати скажімо 350 людей краще ніж 35, але враховуючи кількість комерційних продуктів, як запускаються без належного маркетингового дослідження, навіть 35 респондентів, які мали бажання взаємодіяти з прототипом і залишити свій відгук - це хороший результат. Для більш масштабного дослідження раціональніше робити MVP.

85 % респондентів згодні, що місце розташування житла - одна з ключових характеристик, яка враховується при пошуку. 100 % респондентів підтвердило, що доступність ключових для них об’єктів інфраструктури відіграє ключову роль при виборі житла. Це дуже хороший знак, так як це одна із ключових проблем, яку адресує рекомендаційна система. Адже є багато сайтів нерухомості, але більшість із них не зосереджено на тих об’єктах інфраструктури, які важливі для кожного конкретного користувача, адже потреби у всіх різні. 86 % респондентів згодні, що ізохрони дають наглядне візуальне представлення доступності ключових об'єктів інфраструктури від різних пропозицій житла. Це також досить хороший результат, так як ми не часто зустрічаємо ізохрони у повсякденному житті, але як виявилось, для більшості респондентів - це досить інтуїтивна і зрозуміла концепція. 100 % респондентів погодилось, що при перегляді пропозиції житла, інформація про маршрут до кожної точки інтересу і час - це дуже корисна інформація. Цей функціонал дає змогу перейти від загальних рекомендацій, що у даному полігоні

знаходяться об’єкти, які можуть зацікавити, до конкретних пропозицій, з конкретними величинами, які можна порівнювати між собою. Також переважна більшість респондентів погодилась, що кластеризація - це зручний інструмент, який допомагає швидко оцінити кількість пропозицій на ринку в тій чи іншій частині міста.

Загалом 71 % респондентів зазначили, що не користувались сервісами пошуку житла, в яких враховується доступність об’єктів інфраструктури. 14 % зазначили, що користувались такими сервісами, але функціонал у них був досить обмежений. Також більше 80 % респондентів зазначили, що не користувались і не бачили схожих сервісів на зарубіжних ринках. Звичайно це не означає, що таких сервісів немає. Швидше це говорить, про незначну їх поширеність і колосальні резерви у покращенні клієнтського досвіду для таких платформ. Хорошим прикладом, на мою думку, є французький сайт пошуку житла www.seloger.com. При виборі конкретного варіанту житла вони відмічають найближчі булочні красивими іконками з французьким багетом. Ось наскільки вони піклуються про задоволення потреб своїх користувачів і розуміють свою цільову аудиторію.

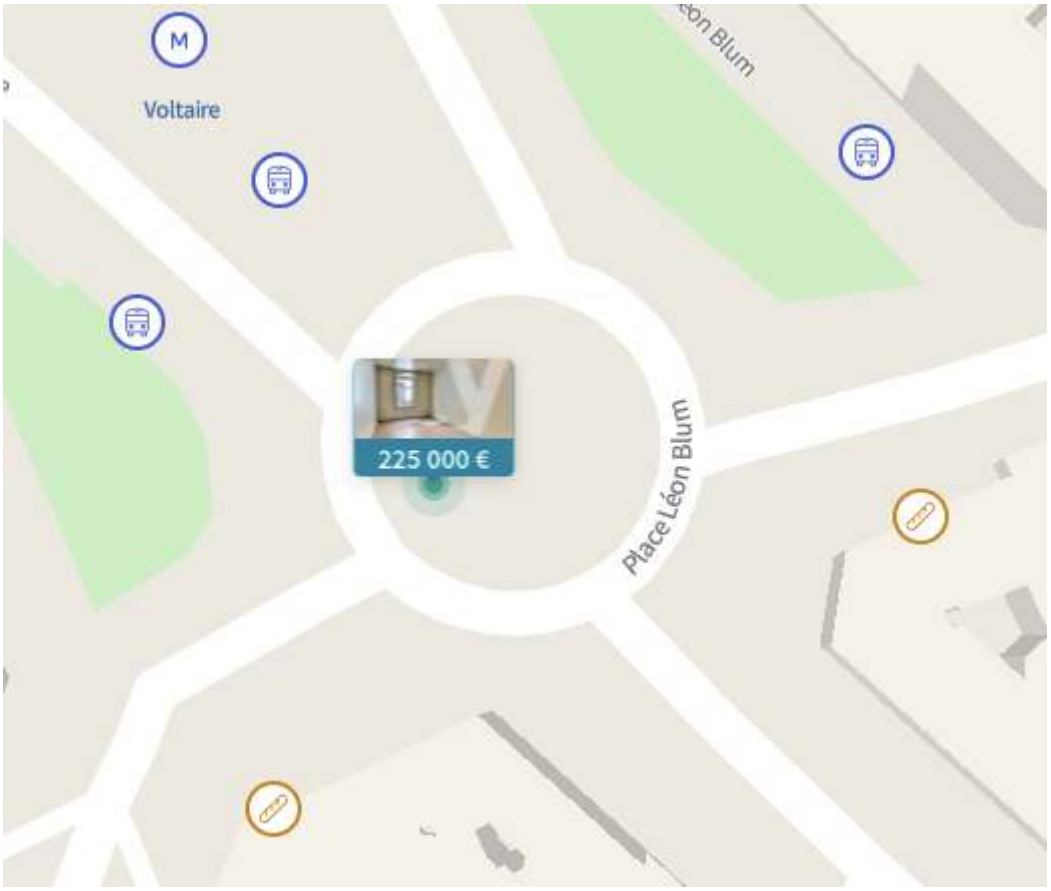


Рисунок. Приклад рекомендаційної системи, що використовує об’єкти інфраструктури.

Крім питань з чітким вибором, в анкеті також були присутні опціональні питання з розгорнутою відповіддю. Респоденти зазначили наступні аспекти, які їм сподобались:

●Мені сподобалась сама ідея. Кожен хоче мати інфраструктуру коло свого дому, але запити на цю інфраструктуру у кожного різні, а тут є можливість вказати, що саме тебе цікавить.

●Простота

●Одночасний розрахунок часу до різних точок, підсвічений загальний "радіус" доступності відповідно до типу пересування

●Крім пунктів, що були вище в питаннях - профіль маршруту - машиною, пішки, велосипедом - дуже зручно. Також подобається можливість задати ліміт по часу. Непоганий текстовий пошук

- швидкодія

Як видно з відгуків - майже всі аспекти, які детально розглядались у дипломній роботі, а саме: швидкодія, геокодування, ізохрони,.. отримали позитивний користувацький відгук, а значить дійсно являються ключовими атрибутами рекомендаційної системи.

Також є аспекти, які на думку респондентів варто покращити для того щоб продукт був готовий до комерційного релізу:

- все ок, тільки цікаво, як воно буде, коли люди захочуть натикати 5, 10, 20 і т.д. цих точок
- мобільна версія, фільтри по типу нерухомості і ціні, тип маршруту - громадський транспорт (метро для Києва)
- можливість вибору для однієї точки декількох варіантів пересування

Із неочевидних, для мене, побажань - вибір для однієї точки декількох варіантів пересування. Це дійсно може бути корисною функцією.

Аналіз виконання поставлених задач

При постановці задачі, для того, щоб можна було співставити результат пророблені дипломної роботи з очікуваним результатом, було розроблено 3 різні сценарії поведінки користувачів.

Перший досить простий. Потрібно знайти житло для першокурсниці, щоб вона змогла за 15 - 20 хвилин дійти пішки до університету.

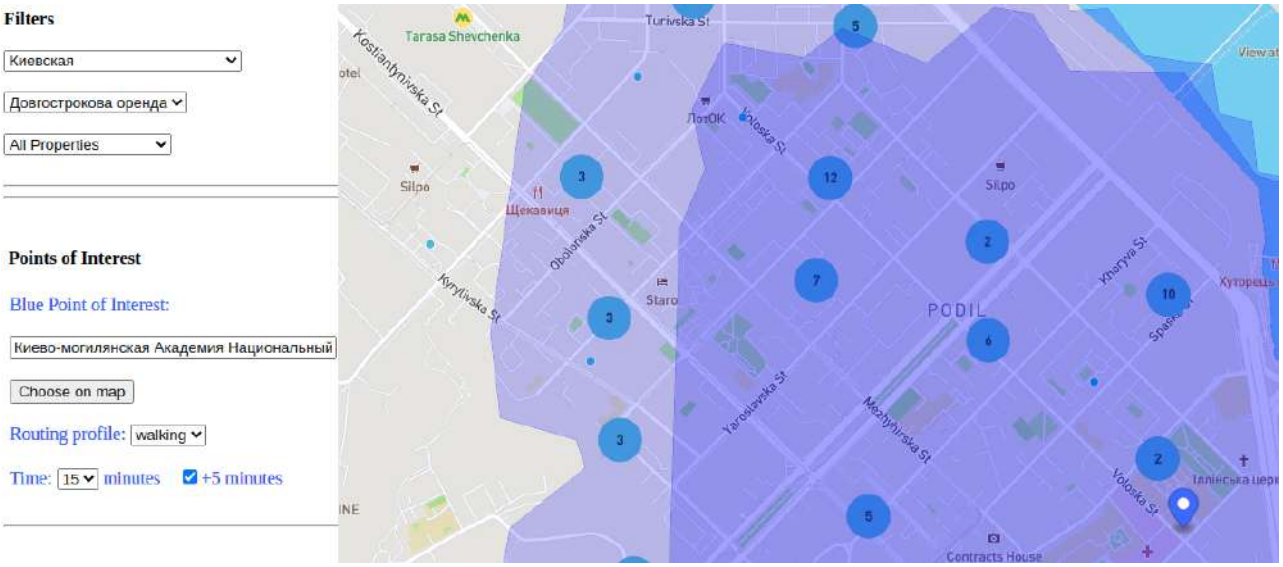


Рисунок. Ізохрон з двома межами 15 і 20 хвилин пішки.

Для того, щоб знайти студентці підходяще житло, я вибрав у фільтрах Київську область та довгострокову оренду. Після чого в полі текстового пошуку почав вводити “Києво-мо...” і вибрав з підказок навчальний заклад. Моя точка інтересу з’явилась на мапі. Потім я вибрав, що хочу пересуватись пішки, і вказав час - 15 хвилин, плюс ще 5 хвилин. Синій полігон з меншою прозорістю покриває житло, від якого можна дістатись за 15 хв, а більш прозорий - житло від якого можна дістатись до університету за 15 - 20 хвилин. Отже з рисунку видно, що перший сценарій покривається архітектурним рішенням і навіть повністю реалізовано у прототипі.

Другий сценарій трохи складніший. Чоловік хоче добратися до місця роботи менш ніж за 20 хвилин на авто. При цьому варто враховувати трафік. Архітектурно ця задача повністю покрита, але в прототипі не реалізована повністю, так як це б збільшило час необхідний на розробку прототипу і сильно ускладнило інтерфейси. Прототип було зроблено максимально простим, для того, щоб не заплутувати респондентів при опитуванні. До прикладу, якщо враховувати трафік, то він сильно залежить від днів тижня а також годин, тому як мінімум потрібно було б розробити інтерфейс для вводу цих параметрів для точок інтересу. Крім того у прототипі був використаний провайдер, який має досить обмежену

інформацію про трафік. Загалом технічно це можливо реалізувати, є провайдери, які дають досить точні результати, наприклад Google Maps, але зі згаданих вище причин, в прототипі це не було реалізовано. Тим не менш, навіть у прототипі, ми можемо зазначити будь яку адресу, як місце роботи, вибрати тип транспорту - авто, а також зазначити час - 20 хв. Більше того, коли ми натиснемо на конкретне житло, то побачимо маршрут до місця роботи, що також являється корисною інформацією.

Наступний сценарій найскладніший. Він також частково реалізований у прототипі, ми звернемо увагу на ці моменти, як їх можливо реалізувати, і наскільки це складно. Суть третього сценарію в тому, що молода сім’я шукає житло. Є три члени сім’ї. Школяр хоче за 15 хв дійти до школи, а чоловік хоче за 40 хвилин доїхати до роботи, при цьому ще й підвезти жінку до її роботи. В прототипі можна вибрати лише дві точки. Таке рішення було прийняте, тому що двох точок достатньо щоб показати найпростіший сценарій, коли потрібно врахувати інтереси по декільком точкам. Реалізовано це в прототипі через перетин двох ізохронів. Відповідно об’єкти нерухомості, які потрапляють у перетин - задовольняють обом вимогам. Технічно не складно зробити довільну кількість точок інтересу, але для прототипу це була б надлишкова складність. Для того щоб задовольнити вимогу “підвезти жінку”, у прототипі можна було б побудувати маршрути не тільки від об’єкту нерухомості до точок інтересу, а й між точками інтересу. Що насправді досить логічно. З технічної точки зору ця задача нічим не відрізняється від задачі побудувати маршрут від об’єкта нерухомості до точки інтересу. Тому з технічної точки зори реалізувати всі вимоги зазначені в сценарії є абсолютно досяжним і вже реалізований функціонал у прототипі є тому підтвердженням.

У даній роботі ми мали на меті дослідити можливість створення платформи, що при пошуку житла орієнтується в першу чергу не на розміщення апартаментів у певному районі, а на доступність об’єктів інфраструктури, які цікавлять

користувача. Розробити прототип. Оцінити економічну доцільність, та практичну цінність такої системи. Поставлені задачі були виконані.

Список літератури

1. Bourke, Paul. “CONREC A Contouring Subroutine.” *Byte. The small systems journal*, 1987.
2. “Browser support.” *Maps and location for developers*, https://docs.mapbox.com/help/troubleshooting/mapbox-browser-support/. Accessed 1 May 2021.
3. Carswell, James, et al. *International Symposium on Web and Wireless Geographical Information Systems*. Berlin, Springer, 2009.
4. “Computer vision approaches for big geo-spatial data: quality assessment of raster tiled web maps for smart city solutions.” *7th International Conference on Cartography and GIS*, 2018, pp. 296-305.
5. Dennis, Dennis. “Creating isochrone catchments from a (distance) matrix.” *Medium - Where good ideas find you*, 18 December 2017, https://medium.com/geolytix/creating-isochrone-catchments-from-a-distance-matrix-15f39e436d09. Accessed 2021.
6. Dibbelt, Julian, et al. “Customizable Contraction Hierarchies.” *ACM Journal of Experimental Algorithmics*, vol. 21, 2016, pp. 1-49.
7. Dijkstra, Edsger. “A Note on Two Problems in Connexion with Graphs.” *Numerische Mathematlk*, 1959, pp. 269-271.
8. Dines, L. L. “On Convexity.” *The American Mathematical Monthly*, vol. 45, no. 4, 1938, pp. 199-209.

9. DOM.RIA.com. *Життя в новобудові: що і як шукають українці*. 2019, https://dom.ria.com/uk/news/zhizn-v-novostrojke-chto-i-kak-ischut-ukrainczy-245206.html. Accessed 2021.
10. Edelsbrunner, H., et al. “On the shape of a set of points in the plane.” *IEEE Transactions on Information Theory*, vol. 29, no. 4, 1983, pp. 551 - 559.
11. Galton, Francis. “On the Construction of Isochronic Passage-Charts.” *Proceedings of the Royal Geographical Society and Monthly Record of Geography*, vol. 3, no. 11, 1881, pp. 657-658.
12. Goldberg, Andrew, and Craig Silverstein. *Implementations of Dijkstra’s Algorithm Based on Multi-Level Buckets. Network Optimization. Lecture Notes in Economics and Mathematical Systems*. vol. 450, Berlin, Heidelberg., Springer, 1997.
13. Mapbox. “Leaflet.” *Maps and location for developers*, https://docs.mapbox.com/help/glossary/leaflet/. Accessed 2021.
14. Nebel, Paul. “Dynamic Server-Side Clustering for Large Datasets.” *Geovation Tech Blog*, 30 January 2018, https://geovation.github.io/dynamic-server-side-geo-clustering. Accessed 2021.
15. Nolde, Nils. “Open Source Routing Engines And Algorithms – An Overview.” 4 December 2020, https://gis-ops.com/open-source-routing-engines-and-algorithms-an-overview/. Accessed 2021.
16. “OLX Partner API.” https://developer.olx.ua/api/doc. Accessed 2021.

17. “RIA.com для разработчиков.” https://developers.ria.com/dom_ria. Accessed 2021.
18. Sample, John T., and Elias Loup. *Tile-Based Geospatial Information Systems*. Berlin/Heidelberg, Germany, Springer, 2010.
19. “Smart homes market - growth, trends, covid-19 impact, and forecasts (2021 - 2026).” 2020, https://www.mordorintelligence.com/industry-reports/global-smart-homes-market-industry. Accessed 2020.
20. Stefanakis, Emmanuel. “Map Tiles and Cached Map Services.” 5 December 2015, https://gogeomatics.ca/map-tiles-and-cached-map-services/. Accessed 2021.

Додаток А. Програмний код серверної частини

File package.json

```
{
 "name": "map_poc",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "start": "node index.js",
   "start-dev": "npx nodemon index.js"
 },
 "keywords": [],
 "author": "",
 "license": "ISC",
 "dependencies": {
   "@koa/router": "^10.0.0",
   "koa": "^2.13.1",
   "koa-better-http-proxy": "^0.2.9",
   "koa-static": "^5.0.0"
 },
 "devDependencies": {
   "eslint": "^7.24.0",
   "eslint-config-airbnb-base": "^14.2.1",
   "eslint-plugin-import": "^2.22.1"
 }
}
```

File index.js

```
const Koa = require('koa');
const Router = require('@koa/router');
const proxy = require('koa-better-http-proxy');


const app = new Koa();
const router = new Router();
router
```

```
  .get('/node/searchEngine/map', proxy('dom.ria.com', { https: true }))
  .get('/dom/info/:id', proxy('developers.ria.com', { https: true }))
  .get(/./, require('koa-static')('public'));

app.use(router.routes());

const port = process.env.PORT || 3030;
app.listen(port);
console.log(`http://localhost:${port}`);
```

Додаток В. Верстка

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="utf-8">
 <title>Property Search Map with Isochrone PoC</title>
 <meta name="viewport"
content="initial-scale=1,maximum-scale=1,user-scalable=no">

 <script src="https://api.mapbox.com/mapbox-gl-js/v2.2.0/mapbox-gl.js"></script>
 <link href="https://api.mapbox.com/mapbox-gl-js/v2.2.0/mapbox-gl.css"
rel="stylesheet">

 <script
src="https://api.mapbox.com/mapbox-gl-js/plugins/mapbox-gl-directions/v4.1.0/mapbox-gl-directions.js"></script>

 <link rel="stylesheet"
href="https://api.mapbox.com/mapbox-gl-js/plugins/mapbox-gl-directions/v4.1.0/mapbox-gl-directions.css"       type="text/css">

 <script
src="https://api.mapbox.com/mapbox-gl-js/plugins/mapbox-gl-geocoder/v4.7.0/mapbox-gl-geocoder.min.js"></script>

 <link rel="stylesheet"
href="https://api.mapbox.com/mapbox-gl-js/plugins/mapbox-gl-geocoder/v4.7.0/mapbox-gl-geocoder.css"       type="text/css">

 <script
src="//cdnjs.cloudflare.com/ajax/libs/numeral.js/2.0.6/numeral.min.js"></script>

 <style>
     body {
         margin: 0;
         padding: 0;
```

```
        }
        #map {
            position: absolute;
            top: 0;
            bottom: 0;
            left: 300px;
            width: 100%;
        }
        .mapboxgl-ctrl-geocoder.mapboxgl-ctrl {
            margin-left: 0;
        }
        .mapboxgl-ctrl-geocoder--input {
            padding-right: 0;
        }
        .mapboxgl-ctrl-geocoder--icon.mapboxgl-ctrl-geocoder--icon-search {
            display: none;
        }
    </style>
</head>
<body>
<div style="display: inline-block; margin-left: 5px">
    <br><b> Filters</b><br><br>
    <label>
      <select id="stateId" onchange="updatePropertiesData(); updateMapCenter()">
        <option value="1">Винницкая</option>
        <option value="11">Днепропетровская</option>
        <option value="13">Донецкая</option>
        <option value="2">Житомирская</option>
        <option value="14">Запорожская</option>
        <option value="15">Ивано-Франковская</option>
        <option value="10" selected>Киевская</option>
        <option value="16">Кировоградская</option>
        <option value="17">Луганская</option>
        <option value="18">Луцкая</option>
        <option value="5">Львовская</option>
        <option value="19">Николаевская</option>
```

```
    <option value="12">Одесская</option>
    <option value="20">Полтавская</option>
    <option value="21">Автономная республика Крым</option>
    <option value="9">Ровенская</option>
    <option value="8">Сумская</option>
    <option value="3">Тернопольская</option>
    <option value="22">Ужгородская</option>
    <option value="7">Харьковская</option>
    <option value="23">Херсонская</option>
    <option value="4">Хмельницкая</option>
    <option value="24">Черкасская</option>
    <option value="6">Черниговская</option>
    <option value="25">Черновицкая</option>
  </select>
</label><br><br>
<label>
  <select id="operationType" onchange="updatePropertiesData()">
    <option value="1" selected>Продаж</option>
    <option value="3">Довгострокова оренда</option>
    <option value="4">Оренда подобова</option>
  </select>
</label><br><br>
<label>
  <select id="inspected" onchange="updatePropertiesData()">
    <option value="0">All Properties</option>
    <option value="1">Only verified Position</option>
  </select>
</label><br><br>
<hr>
<hr>
<br><br>
 <b>Points of Interest</b><br><br>
<div style="display: inline-block; color: blue; padding: 5px">
  <span>Blue Point of Interest:</span><br><br>
  <input id="searchTextField" type="text" size="33" placeholder="search">
  <br><br>
```

```
    <div id="geocoder"></div>
    <button onclick="chooseOnMap()">Choose on map</button>
    <br>
    <br>
    <label>Routing profile:
      <select id="routingProfile1" onchange="updateIsochrone()">
        <option value="walking">walking</option>
        <option value="cycling" selected>cycling</option>
        <option value="driving">driving</option>
      </select>
    </label><br><br>
    <label>Time:
      <select id="time" onchange="updateIsochrone()">
        <option value="5">5</option>
        <option value="10" selected>10</option>
        <option value="15">15</option>
        <option value="20">20</option>
        <option value="25">25</option>
        <option value="30">30</option>
        <option value="35">35</option>
        <option value="40">40</option>
      </select>
      minutes
    </label>  
    <label><input type="checkbox" id="secondLine1" onchange="updateIsochrone()">+5
minutes</label>
 </div>
 <br>
 <br>
 <hr>
 <br>
 <div style="display: inline-block; color: green; padding: 5px">
    <span>Green Point of Interest:</span><br><br>
    <button onclick="chooseSecondPoint()">Choose on map</button>
    <br><br>
    <label>Routing profile:
```

```
      <select id="routingProfile2" onchange="updateSecondIsochrone()">
        <option value="walking">walking</option>
        <option value="cycling">cycling</option>
        <option value="driving" selected>driving</option>
      </select>
    </label><br><br>
    <label>Time:
      <select id="time2" onchange="updateSecondIsochrone()">
        <option value="5">5</option>
        <option value="10" selected>10</option>
        <option value="15">15</option>
        <option value="20">20</option>
        <option value="25">25</option>
        <option value="30">30</option>
        <option value="35">35</option>
        <option value="40">40</option>
      </select>
      minutes
    </label>
      
    <label>
      <input type="checkbox" id="secondLine2"
onchange="updateSecondIsochrone()">+5 minutes
    </label>
    <br>
  </div>
</div>
<div id="map"></div>
<script src="states.js"></script>
<script src="index.js"></script>
<script

src="//maps.googleapis.com/maps/api/js?callback=addGeocoder&v=3&libraries=places&
key=AIzaSyCcFeJm50SYHOl56jwCRvUd3xWmP8DSprA"></script>
</body>
</html>
```

Додаток С. Програмний код клієнтської частини

```
window.stateIdToLngLatPairs = {
 1: [28.4678193206055200, 49.2326765304214700],
 2: [28.6662493452319500, 50.2681374406948200],
 3: [25.5977455528101500, 49.5491069645468100],
 4: [26.9953792102950900, 49.4158079577649500],
 5: [24.0113364012023350, 49.8300039411982140],
 6: [31.2985490111097700, 51.4971440021006000],
 7: [36.2788473894288000, 49.9867804575263150],
 8: [34.8148187798800850, 50.9028319497817100],
 9: [26.2473261515938700, 50.6108789885385950],
 10: [30.5310709814418040, 50.4019884552972300],
 11: [34.9956263076269800, 48.4599135158961700],
 12: [30.7116734065825840, 46.4659725044436060],
 13: [37.7619610663489800, 47.9884945203922440],
 14: [35.1758613022437500, 47.8525940261186100],
 15: [24.7154675464181570, 48.9093534629659600],
 16: [32.2477965267345200, 48.5197149926495300],
 17: [39.3569146306047200, 48.5800679376263100],
 18: [25.3302555915996240, 50.7393024658362600],
 19: [31.9874560105784270, 46.9378700326313400],
 20: [34.5580087272677900, 49.6022515219643700],
 21: [34.0903751901182660, 44.9642010542266000],
 22: [22.2911878013622000, 48.6229244470405460],
 23: [32.6231661408154140, 46.6483175589648600],
 24: [32.0533114888137300, 49.4314779471190850],
 25: [25.9339237021311750, 48.3213519985855100],
};


const accessToken = '';
const { mapboxgl, MapboxGeocoder } = window;
mapboxgl.accessToken = accessToken;
const zoom = 11;
const map = new window.mapboxgl.Map({
 container: 'map', // container id
```

```
 style: 'mapbox://styles/mapbox/streets-v11', // style URL
 center: window.stateIdToLngLatPairs[10],
 zoom, // starting zoom
});
const marker1 = new mapboxgl.Marker({ color: 'blue'
}).setLngLat(window.stateIdToLngLatPairs[10])
 .addTo(map);
let marker2;
const markers = [marker1];
const opacity = 0.2;

function getInputValue(name) {
 return document.getElementById(name).value;
}

function getIsochroneApiUrl({
 routingProfile, lng, lat, time, id,
}) {
 const longitudeLatitude = encodeURIComponent([lng, lat].join());
 const times = [time];
 const element = document.getElementById(`secondLine${id}`);
 if (element && element.checked) times.push(Number(time) + 5);
 const contourMinutes = encodeURIComponent(times.join());
 const queryStr =
`contours_minutes=${contourMinutes}&polygons=true&access_token=${accessToken}`;
 return
`https://api.mapbox.com/isochrone/v1/mapbox/${routingProfile}/${longitudeLatitude
}?${queryStr}`;
}

const getFirstIsochroneUrl = () => getIsochroneApiUrl({
 routingProfile: getInputValue('routingProfile1'),
 time: getInputValue('time'),
 ...marker1.getLngLat(),
 id: 1,
});
```

```
function updateIsochrone() {
 map.getSource('isochrone').setData(getFirstIsochroneUrl());
}

function updateMarker(lng, lat) {
 marker1.setLngLat([lng, lat]);
}

function updateMarkerAndIsochrone(lng, lat) {
 updateMarker(lng, lat);
 updateIsochrone();
}

// eslint-disable-next-line no-unused-vars
function chooseOnMap() {
 map.once('click', ({ lngLat: { lat, lng } }) => {
   updateMarkerAndIsochrone(lng, lat);
 });
}

// eslint-disable-next-line no-unused-vars
function updateSecondIsochrone() {
 if (!marker2) return;
 const { lng, lat } = marker2.getLngLat();
 const apiUrl = getIsochroneApiUrl({
   routingProfile: getInputValue('routingProfile2'),
   time: getInputValue('time2'),
   lng,
   lat,
   id: 2,
 });
 map.getSource('isochrone2').setData(apiUrl);
}

// eslint-disable-next-line no-unused-vars
function chooseSecondPoint() {
```

```
 map.once('click', ({ lngLat: { lat, lng } }) => {
   const apiUrl = getIsochroneApiUrl({
     routingProfile: getInputValue('routingProfile2'),
     time: getInputValue('time2'),
     lng,
     lat,
     id: 2,
   });
   if (marker2) {
     marker2.setLngLat([lng, lat]);
     map.getSource('isochrone2').setData(apiUrl);
     return;
   }
   marker2 = new mapboxgl.Marker({ color: 'green' }).setLngLat([lng,
lat]).addTo(map);
   markers.push(marker2);
   // secondIsochroneCenter = [lng, lat];
   map.addSource('isochrone2', { type: 'geojson', data: apiUrl });
   // Add a new layer to visualize the polygon.
   map.addLayer({
     id: 'isochrone2',
     type: 'fill',
     source: 'isochrone2', // reference the data source
     layout: {},
     paint: { 'fill-color': 'green', 'fill-opacity': opacity },
   });
 });
}

async function getPropertiesData() {
 let api = '/node/searchEngine/map/?category=1&secondary=1';
 api += `&operation_type=${getInputValue('operationType')}`;
 api += `&state_id=${getInputValue('stateId')}`;
 if (Number(getInputValue('inspected'))) api += '&inspected=1';
 const res = await fetch(api);
 const { items } = await res.json();
```

```
 const features = items.map((item) => ({
   type: 'Feature',
   geometry: { type: 'Point', coordinates: [item.longitude, item.latitude] },
   properties: { id: item.realty_id },
 }));
 return { type: 'FeatureCollection', features };
}

async function getRealtySource() {
 const data = await getPropertiesData();
 return {
   type: 'geojson',
   data,
   cluster: true,
   clusterMaxZoom: 14, // Max zoom to cluster points on
   clusterRadius: 50, // Radius of each cluster when clustering points (defaults
to 50)
 };
}

// eslint-disable-next-line no-unused-vars
async function updatePropertiesData() {
 map.getSource('properties').setData(await getPropertiesData());
}

// eslint-disable-next-line no-unused-vars
function updateMapCenter() {
 const center = window.stateIdToLngLatPairs[getInputValue('stateId')];
 map.setCenter(center);
 map.setZoom(zoom);
}

function makeClusterLayer() {
 return {
   id: 'clusters',
   type: 'circle',
```

```
    source: 'properties',
    filter: ['has', 'point_count'],
    paint: {
      // Use step expressions
(https://docs.mapbox.com/mapbox-gl-js/style-spec/#expressions-step)
      // with three steps to implement three types of circles:
      //   * Blue, 20px circles when point count is less than 100
      //   * Yellow, 30px circles when point count is between 100 and 750
      //   * Pink, 40px circles when point count is greater than or equal to 750
      'circle-color': ['step', ['get', 'point_count'], '#51bbd6', 100, '#f1f075',
750, '#f28cb1'],
      'circle-radius': ['step', ['get', 'point_count'], 20, 100, 30, 750, 40],
    },
  };
}

function makeCountInClasterLayer() {
  return {
    id: 'cluster-count',
    type: 'symbol',
    source: 'properties',
    filter: ['has', 'point_count'],
    layout: {
      'text-field': '{point_count_abbreviated}',
      'text-font': ['DIN Offc Pro Medium', 'Arial Unicode MS Bold'],
      'text-size': 12,
    },
  };
}

function makeUnclusteredPointLayer() {
  return {
    id: 'unclustered-point',
    type: 'circle',
    source: 'properties',
    filter: ['!', ['has', 'point_count']],
```

```
    paint: {
      'circle-color': '#11b4da',
      'circle-radius': 4,
      'circle-stroke-width': 1,
      'circle-stroke-color': '#fff',
    },
  };
}

function zoomClusterOnClick() {
  map.on('click', 'clusters', (e) => {
    const features = map.queryRenderedFeatures(e.point, { layers: ['clusters'] });
    const clusterId = features[0].properties.cluster_id;
    map.getSource('properties').getClusterExpansionZoom(clusterId, (err, zoom) =>
{
      if (err) return;
      map.easeTo({ center: features[0].geometry.coordinates, zoom });
    });
  });
}

function changeMousePointerOnclusterHover() {
  map.on('mouseenter', 'clusters', () => {
    map.getCanvas().style.cursor = 'pointer';
  });
  map.on('mouseleave', 'clusters', () => {
    map.getCanvas().style.cursor = '';
  });
}

function removeRoutesOnPopUpClose(popup) {
  popup.on('close', () => {
    map.removeLayer('route1');
    map.removeSource('route1');
    if (marker2) {
      map.removeLayer('route2');
```

```
      map.removeSource('route2');
    }
  });
}

function showPopUp(feature, poiTimes) {
  const coordinates = feature.geometry.coordinates.slice();
  const realtyId = feature.properties.id;
  fetch(`/dom/info/${realtyId}?api_key=`)
    .then((res) => res.json())
    .then((realty) => {
      const src =
`https://cdn.riastatic.com/photos/${realty.main_photo.replace(/\.(.{3})/,
        'fl.$1')}`;
      const img = `<img src="${src}" height="115" width="155" alt="property
image"/>`;
      const href = `https://dom.ria.com/uk/realty-${realtyId}.html`;
      let html = `<a href="${href}" target="_blank">${img}</a>`;
      html += `<br><b>${realty.priceArr[1]} $</b>, ${realty.rooms_count} rooms`;
      html += `   <a href="${href}" target="_blank">view -></a>`;
      poiTimes.forEach(({ duration, distance, routingProfile }, idx) => {
        const durationMin = Math.round(duration / 60);
        const distanceRounded = Math.round(distance / 100) / 10;
        // eslint-disable-next-line no-underscore-dangle
        const color = markers[idx]._color;
        html += `<br>
          <span style="color: ${color}">
            ${durationMin} min. ${routingProfile} ${distanceRounded} km
          </span>`;
      });
      const popup = new mapboxgl.Popup();
      popup.setLngLat(coordinates).setHTML(html).addTo(map);
      removeRoutesOnPopUpClose(popup);
    });
}
```

```
function showRoute(feature, marker, id) {
 const { lng, lat } = marker.getLngLat();
 const coordinates = feature.geometry.coordinates.slice();
 const routingProfile = getInputValue(`routingProfile${id}`);
 let directionApi =
`https://api.mapbox.com/directions/v5/mapbox/${routingProfile}`;
 if (routingProfile === 'driving') directionApi += '-traffic';
 directionApi += '/';
 directionApi += encodeURIComponent([coordinates.join(), [lng,
lat].join()].join(';'));
 directionApi +=
`?alternatives=false&geometries=geojson&steps=false&access_token=${accessToken}`;
 return fetch(directionApi).then((res) => res.json()).then((json) => {
   const { geometry, duration, distance } = json.routes[0];
   map.addSource(`route${id}`, { type: 'geojson', data: geometry });
   map.addLayer({
     // eslint-disable-next-line no-underscore-dangle
     id: `route${id}`,
     type: 'line',
     source: `route${id}`,
     // eslint-disable-next-line no-underscore-dangle
     paint: { 'line-color': marker._color, 'line-width': 2 },
   });
   return { duration, distance, routingProfile };
 });
}

function showPopUpOnpointClick() {
 map.on('click', 'unclustered-point', (event) => {
   const feature = event.features[0];
   const promises = [showRoute(feature, marker1, 1)];
   if (marker2) promises.push(showRoute(feature, marker2, 2));
   Promise.all(promises).then((data) => showPopUp(feature, data));
 });
}
```

```
async function drawIsochronePolygon() {
 map.addSource('isochrone', { type: 'geojson', data: getFirstIsochroneUrl() });
 // Add a new layer to visualize the polygon.
 map.addLayer({
   id: 'isochrone',
   type: 'fill',
   source: 'isochrone', // reference the data source
   layout: {},
   paint: { 'fill-color': 'blue', 'fill-opacity': opacity },
 });
}

// eslint-disable-next-line no-unused-vars
function addGeocoder() {
 const input = document.getElementById('searchTextField');
 const gMaps = window.google.maps;
 const autocomplete = new gMaps.places.Autocomplete(input, { country: 'ua' });
 autocomplete.addListener('place_changed', () => {
   const place = autocomplete.getPlace();
   // console.log(place);
   const { geometry: { location } } = place;
   updateMarkerAndIsochrone(location.lng(), location.lat());
 });
}

async function main() {
 map.addSource('properties', await getRealtySource());
 map.addLayer(makeClusterLayer());
 map.addLayer(makeCountInClasterLayer());
 map.addLayer(makeUnclusteredPointLayer());
 zoomClusterOnClick();
 showPopUpOnpointClick();
 changeMousePointerOnclusterHover();
 await drawIsochronePolygon();
}
```

```
main();
```