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

Кафедра інформатики факультету інформатики
Спеціальність „Інженерія програмного забезпечення”

Курсова робота

Сучасні підходи у проектуванні високонавантажених мап для пошуку об’єктів

Науковий керівник:
доц., доктор техн. наук
Глибовець Андрій Миколайович
_____(підпис) “_____” _____ 2020 р.
Виконав студент 1 курсу магістратури:
Жук Максим Анатолійович

Київ 2018

Міністерство освіти і науки України

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики,

к.ф.-м.н., доц. Гороховський С.С._____(підпис)

„____”_____2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

Студенту, Жуку Максиму Анатолійовичу, факультету інформатики 1 курсу

ТЕМА: “Сучасні підходи у проектуванні високонавантажених мап для пошуку об’єктів”

Зміст ТЧ до курсової роботи:

- Індивідуальне завдання
- Календарний план
- Анотація
- Вступ
- Аналіз поставленої задачі.
- Аналіз наявних технологій.
- Проектування архітектури.
- Висновки
- Список використаної літератури

Дата видачі „___” _____ 2020 р.

Керівник _____(підпис)

Завдання отримав _____(підпис)

Тема: Методи приблизного пошуку найближчих сусідів

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу
1.	Отримання завдання на дипломну роботу.	16.10.2019
2.	Огляд технічної літератури за темою роботи.	16.11.2019
3.	Аналіз існуючих досліджень за темою роботи.	14.12.2019
3.	Написання вступу та плану роботи.	11.03.2020
4.	Аналіз можливих підходів до вирішення задачі	15.04.2020
6.	Написання основних розділів роботи.	09.05.2020
7.	Створення слайдів для доповіді та написання доповіді.	11.05.2020
8.	Коригування роботи відповідно до вимог щодо оформлення робіт.	13.05.2020
9.	Остаточне оформлення пояснювальної роботи та слайдів.	16.05.2020
0.	Коригування роботи згідно із зауваженнями керівника.	17.05.2020
1.	Захист курсової роботи	18.05.2020

Студент Жук М. А.

Керівник Глибовець А. М.

“_____”_____

ЗМІСТ

Вступ

Актуальність теми.

Важко уявити сучасне життя без мап. Сьогодні мапи є всюди: комп’ютер, телефон, машина, навіть у сучасному годиннику може бути мапа. А чи задумувались ви, наскільки це технологічно складний інструмент? На мапі можуть знаходитись тисячі, або й навіть десятки тисяч об’єктів, користувач може не знати точну назву об’єкту, який йому потрібен, або може бути кілька об’єктів з однаковою назвою у різних регіонах. І це лише кілька проблем. Ми не беремо до уваги такі глобальні, як збір інформації для побудови мапи, або визначення геолокації користувача.

Мета роботи

У даній курсовій роботі ми розглянемо кілька аспектів пов’язаних з прикладною реалізацією проектів з використанням мап: складність виводу великої кількості об’єктів на мапу, складність пошуку та фільтрації, а також витрати. Для цього буде вибраний реалістичний кейс для побудови високонавантаженої мапи.

Кінцеві артефакти

Виходячи з нашого кейсу, ми складемо перелік функціональних та нефункціональних вимог. Розглянемо складнощі, які з пов’язані з цими вимогами, різні підходи для їх вирішення, та в результаті розробимо архітектурне рішення для імплементації

проекту. Архітектурне рішення буде включати в себе порівняльні таблиці провайдерів мап, контекстну та компоненту діаграму з поясненням використаних патернів та технологій.

Розділ 1. Аналіз поставленої задачі

1.1 Постановка задачі

Для того щоб розгляд технологій і архітектури носив практичний характер і мав прикладну цінність нам перш за все потрібно вибрати реалістичний сценарій для розробки нашої мапи. Основні складнощі при проектуванні і розробці мапи виникають при високих навантаженнях: велика кількість об’єктів на мапі, або велика кількість користувачів. Приклади таких мап часто можна зустріти на сайтах пов’язаних з нерухомістю, наприклад booking.com. Це пов’язано з тим, що при пошуку нерухомості, її положення являється однією з найпріоритетніших характеристик. Іншими словами - нерухомість неможливо порухати. В даній курсовій роботі запропоновано розглянути наступний бізнес сценарій: потрібно створити веб додаток на якому користувачі можуть швидко та зручно знайти будь-який об’єкт нерухомості в межах України для того щоб купити або орендувати його. Давайте розглянемо функціональні та нефункціональні вимоги до нашого продукту.

1.2 Функціональні вимоги

Очевидно що основною функціональною вимогою до нашого продукту буде наявність можливості побачити всі об’єкти нерухомості на карті. За даними одного з ведучих сайтів нерухомості України, на мапі потенційно може бути до 100 000 об’єктів нерухомості. Зрозуміло, що орієнтуватись у такій кількості об’єктів дуже складно, тому необхідно забезпечити зручний пошук,

фільтрацію та навігацію. Потрібен як пошук об’єкту, який користувач хоче купити, наприклад “ЖК Новий дім”, так і пошук місця інтересу. Наприклад користувача може цікавити житло біля парку імені Горького. Крім того користувач може не знати правильної назви об’єкту, або випадково зробити помилку, є об’єкти з англійськими назвами а також синонімічні назви. Узагальнимо вищесказане у короткий і зрозумілий список, для того щоб надалі нам було зручно користуватись ним і робити відсилки.

- на мапі мають відображатись всі об’єкти нерухомості України, які можливо купити або зняти в оренду
- пошук об’єкту для покупки/оренди по назві
- пошук об’єкту інтересу по назві
- автодоповнення при пошуку об’єкта за назвою об’єкту по пешим літерам.
- фільтрація об’єктів(ціна, площа,..)

1.2 Нефункціональні вимоги

Наступним кроком розглянемо нефункціональні вимоги. Нефункціональні вимоги визначають атрибути якості системи. Наприклад: швидкодія, надійність, масштабування, зручність, легкість підтримки, та багато інших атрибутів якості системи.

Так як наша мапа високонавантажена(до 100 000 об’єктів), то одним з основних атрибутів якості буде швидкодія. Для того щоб визначити допустимі норми швидкодії проаналізуємо середньостатистичні показники. HTTP Archive являється одним з найавторитетніших ресурсів, який збирає дані по всьому світі протягом багатьох років, а також агрегує дані з Chrome User

Experience Report. За даними HTTP Archive середня швидкість повного завантаження сторінки 6.5 секунд.

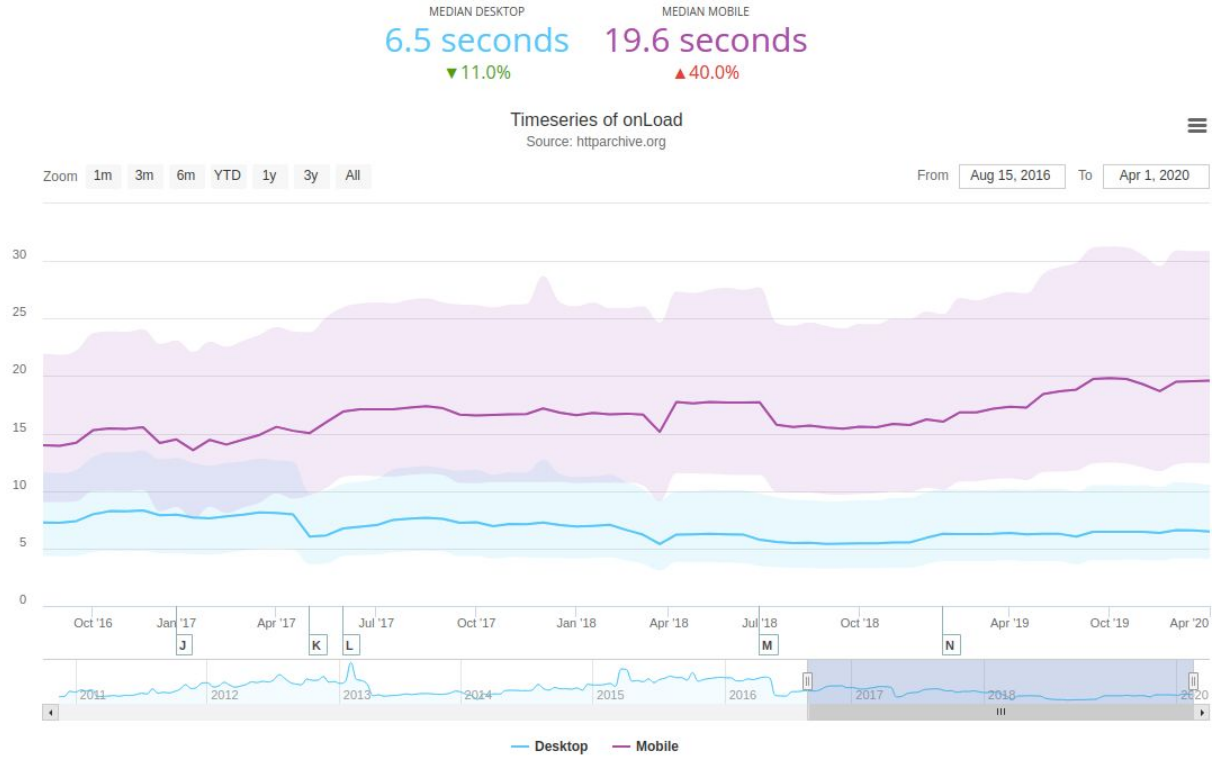


Рисунок 1.2.1 - Швидкість завантаження сторінки

В той же час швидкість завантаження DOM дерева складає 2.8 секунди. Також різні дослідження показують, що половина користувачів не готова чекати на сайт, який завантажується більше 3с. Тому ми візьмемо цей показник, як граничну межу. Швидкість оновлення даних звичайно має бути швидша ніж повне завантаження сторінки, враховуючи об’єм данних 2 секунди може бути прийнятним показником.

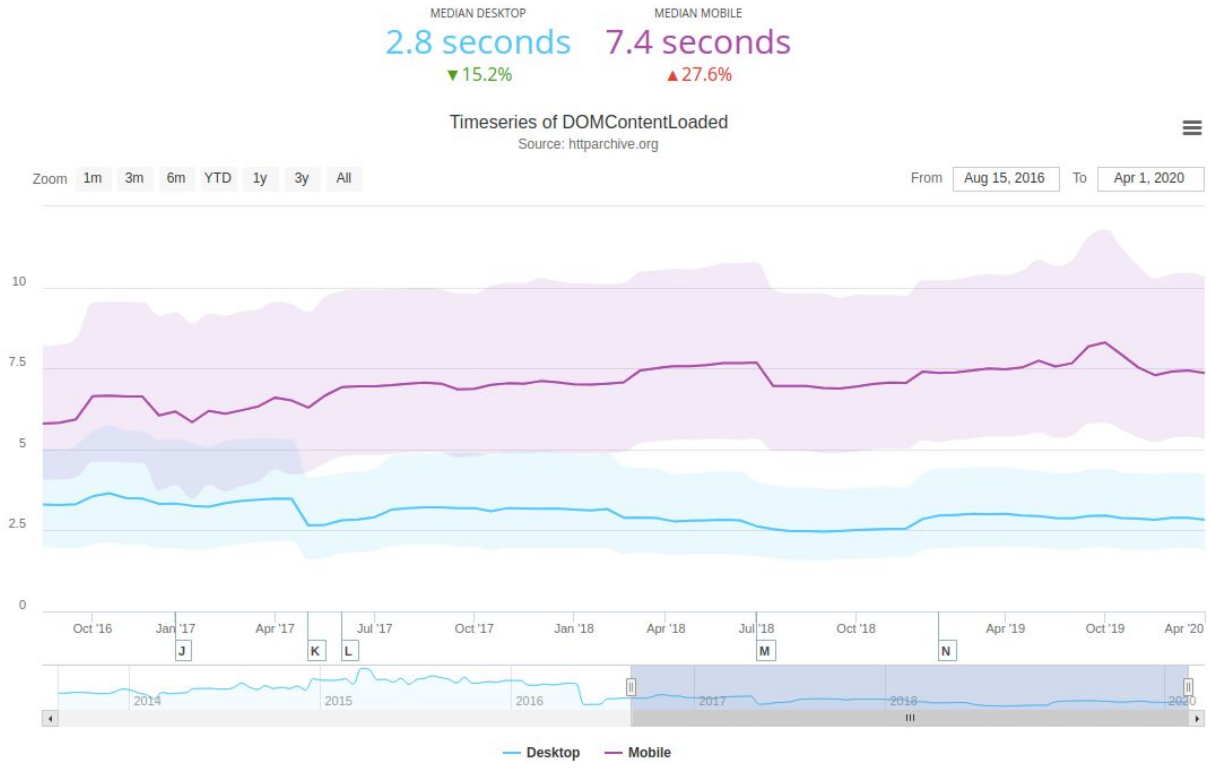


Рисунок 1.2.1 - Швидкість завантаження DOM дерева

Також у наших функціональних вимогах фігурує автодоповнення. До нього також особливі вимоги, так як користувач може друкувати досить швидко і доповнення мають працювати не менш швидко.В середньому людина друкує 40 слів за хвилину, або 200 символів. Що в результаті дає нам: 60 / 200 * 1000 = 300 мс затримки між вводом літер. Це і буде наша гранична межа для системи автодоповнення. Крім того, що автодоповнення мають працювати швидко, вони мають витримувати досить велике навантаження, так як введення кожної нової літери породжує додатковий запит на API. Система має витримувати 1 000 000 сеансів на місяць. Звичайно не всі користувачі будуть одночасно користуватись пошуком через автодоповнення, але так як кожна

літера генерує новий запит, то будемо вважати, що система має бути розрахована так само на 1 000 000 запитів.

Очевидно, що ми не будемо створювати мапу повністю самі, так як це невиправдано дорого і довго. Ми будемо користуватись публічними даними, безкоштовними та платними API для того, щоб втілити проект в життя. Операційні витрати можуть бути дуже суттєвими, враховуючи нашу кількість користувачів і об’єктів на мапі. Ці витрати можуть перевищити потенційний прибуток в рази, що може зробити даний проект недоцільним з точки зору бізнесу. Тому нам варто з самого початку визначити граничну межу для операційних витрат. Припустимо що це буде 2 000 $ в рік. Це достатній бюджет, щоб мати кілька опцій імплементації і враховуючи, що витрати не будуть перевищувати 200$ в місяць - це гарантовано не буде мати катастрофічний вплив на бізнес.

Підсумуємо вищезгадані вимоги:

- мапа має завантажуватись менш, ніж за 3с
- оновлення результатів(фільтрація) має відбуватись менш ніж за 2с
- підсказки автодоповнення мають з’являтись за менш ніж 300мс.
- Система має витримувати 1 000 000 сеансів на місяць.
- API автодоповнення має витримувати до 1 000 000 запитів за на місяць
- витрати на карту не мають перевищувати 2 000 $ в рік.

Розділ 2. Аналіз провайдерів та технологій.

2.1 Клієнтські бібліотеки

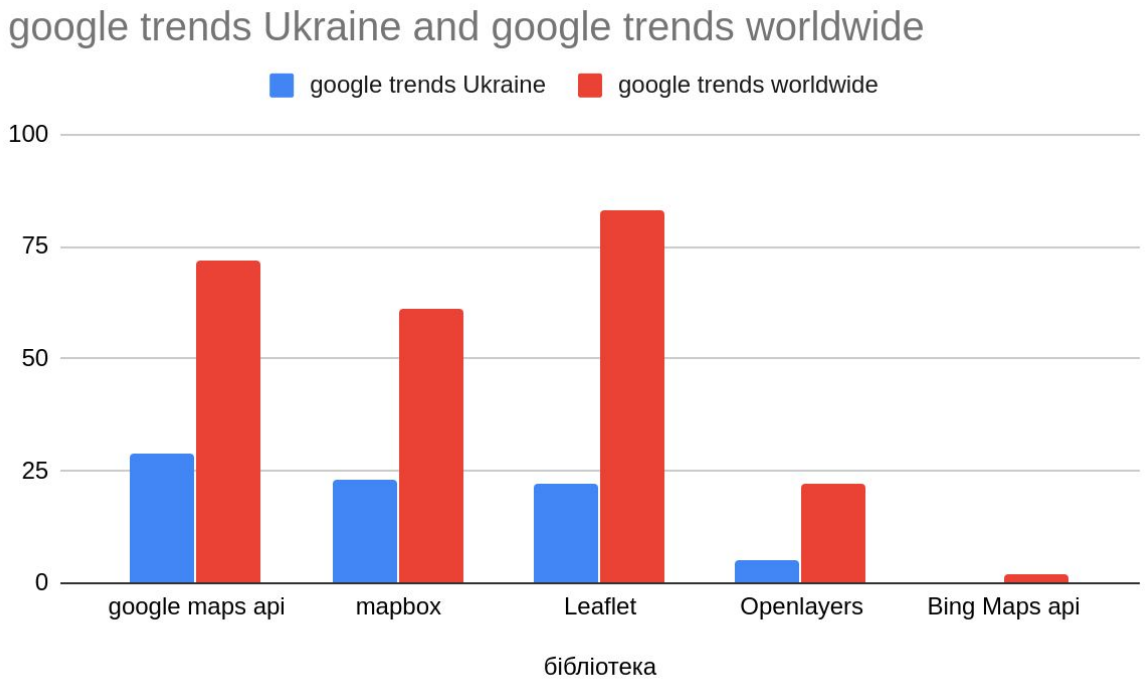
Ми розробляємо веб додаток, тому нас перш за все цікавить основна бібліотека, яка буде забезпечувати відображення мапи, та робити з нею всі маніпуляції. На ринку є досить багато бібліотек [1]:

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

Робити огляд більше десятка бібліотек займе недоцільно багато часу. 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 у світі репозиторій javascript бібліотек
- google trends - аналітика пошукових запитів по всьому світі
- github - веб портал у якому інтегровано безліч інструментів для розробки коду, в тому числі рейтингова система
- npm trends - веб портал, який збирає статистику завантажень бібліотек з npm і відображає порівняльні графіки у розрізі часового періоду.



Діаграма 2.1.1 - Популярність бібліотек відповідно до google trends

Згідно Google Trends найпопулярнішими як в Україні так і світі є Google maps, Mapbox та Leaflet. Причому Google maps дуже

популярна не тільки як платформа для розробки користувацьких карт, а й як самостійний продукт. Це також важливий фактор, адже багато користувачів звикло до інтерфейсів Google maps. Проте в них є один суттєвий мінус, це найдорожче API.

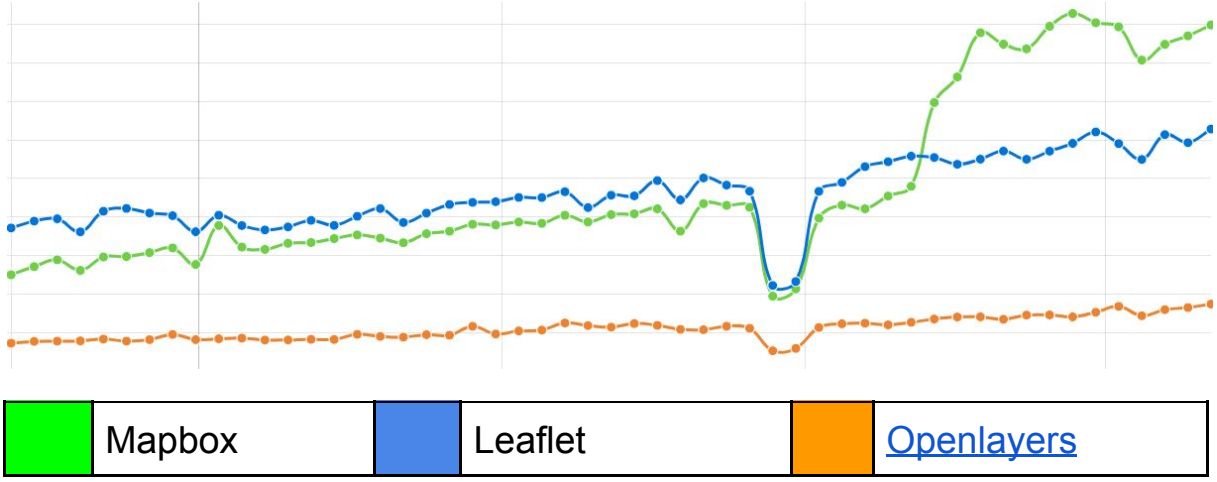
	MONTHLY VOLUME RANGE (Price per MAP LOAD)
0–100,000	100,001–500,000
0.007 USD per each (7.00 USD per 1000)	0.0056 USD per each (5.60 USD per 1000)

Рисунок 2.1.1 - Pricing for Google Maps JavaScript API

По оцінкам користувачів github Leaflet беззаперечний лідер і має на момент написання курсової роботи майже 28 тисяч зірок. Найближчі конкуренти Openlayers та Mapbox мають 7 та 6 тисяч відповідно. Проте слід вважати, що їх з великою ймовірністю міг би обігнати Google, якби використовував Github.

Mapbox завантажують більше 400 тисяч разів за тиждень, тоді як Leaflet 300 тисяч. Це не критична різниця. Leaflet має близько 300 відкритих питань, а Mapbox в 2 рази більше. Проте судячи з розміру бібліотеки(в 10 раз більша) Mapbox має більшу функціональність, тому кількість питань цілком виправдана. На момент написання Leaflet останній раз оновлювався 5 місяців тому, що не є найкращим показником. 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 2.1.1 - 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 2.1.1 - порівняльна таблиця клієнтських бібліотек.

2.2 Підложка мапи

В закінченому вигляді мапа представляє собою певну підложку, де намальовані основні географічні об’єкти, і вся додаткова функціональність, яку ми додали: додаткові об’єкти, елементи управління,..

Перша веб-мапа була представлена компанією Xerox у 1993 році [2]. Це було одне суцільне зображення і при кожній взаємодії з нею завантажувалось абсолютно нове зображення, конкретно для

данного екрану, широти, довготи, масштабу. Такий підхід створює дуже велике навантаження на сервери, мапа завантажується повільно, трафік використовується нещадно, а інтерактивність залишає бажати кращого.

Компанія Google в 2005 році запропонувала підхід, який став революцією на ринку і дефакто стандартом вже більше 15 років. Цей підхід дуже схожий на те, як взаємодіють з великими паперовими картами. Зазвичай вони зігнуті в кілька разів, а можуть бути й різні листки. Тому іноді, щоб переглянути певну область - потрібно зістикувати ці листки. Google запропонував по схожому принципу розбити зображення на плитки(tiles). Ці плитки можна заздалегідь згенерувати [3]. Таким чином при зміщеннях вам лише необхідно підгрузити кілька нових плиток, а не повне зображення мапи, яке ви бачите. Для кожного рівня наближення мапи генерується свій набір плиток.

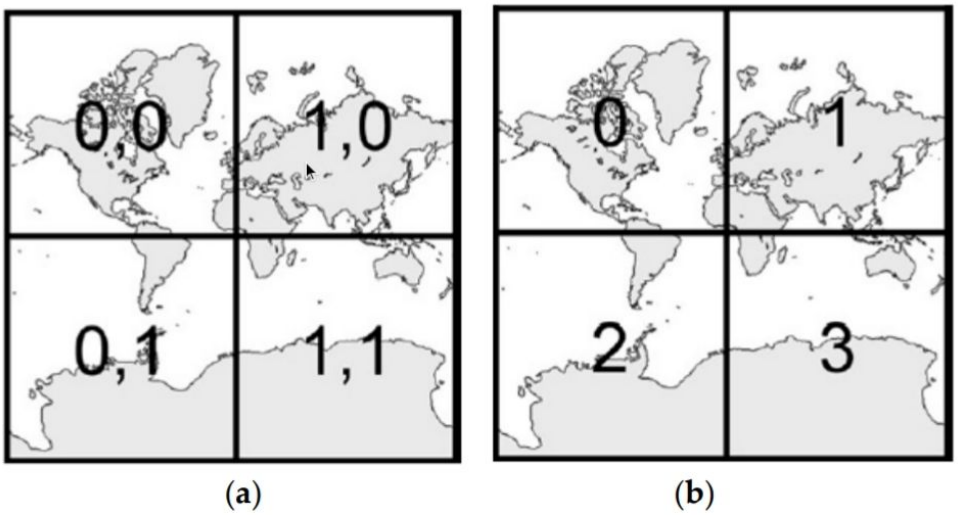


Рисунок 2.2.1 - Схема плиток Google(a) та Bing(b).

Є два типи плиток: растрові і векторні. Різниця між ними та сама, що й між растровими і векторними типами зображень.

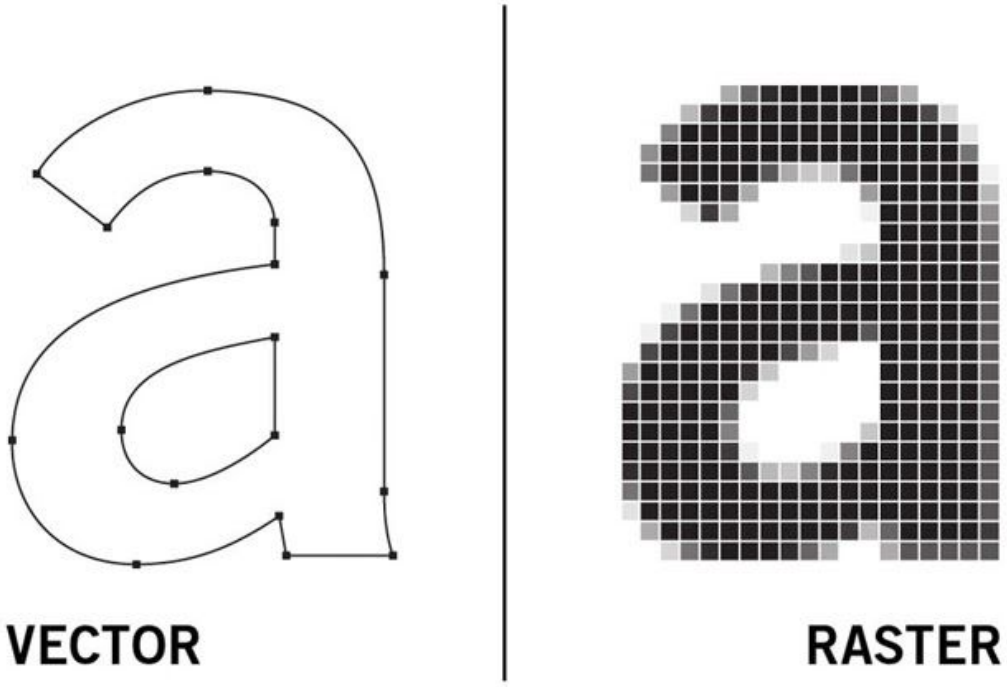


Рисунок 2.2.2. - Різниця між растровими і векторними зображеннями.

До переваг растрових тайлів можна віднести високу швидкість генерації у порівнянні з векторними. Їх можливо навіть генерувати нальоту за кілька секунд і таким чином можливо зекономити на дисковому просторі. Також растрові плитки мають хорошу кросбраузерну підтримку. Але є в них і певні недоліки. При будь-якій зміні на карті їх потрібно повністю генерувати заново. Також з растровою картою важко реалізувати взаємодію [4], тому що з точки зору браузера це лише картинка і вирізнити на ній окремі елементи для взаємодії досить нетривіальна задача.

Векторні плитки були запропоновані компанією Google у 2010 році [5]. Ідея полягала у тому, щоб зберігати на сервері не картинки, а опис об’єктів на мапі: полігони, лінії, точки. А також опис до об’єктів: назви вулиць, міст,.. Найбільш поширений формат

збереження данних - GeoJSON. Він оснований форматі JSON, який на даний момент являється найпопулярнішим форматом передачі даних, через свою простоту та компактність. Ще одним популярним форматом є Mapbox Vector Tile, який використовує бінарний протокол protobuf, а тому ще більш компактний. Однією з основних переваг векторних плиток є можливість динамічно змінювати стилізацію мапи. Це забезпечується тим, що об’єкти та стилі зберігаються у різних файлах.

Варто зазначити, що незважаючи на переваги векторних плиток Google ще й досі повністю не перейшла на них. Мабуть це пов’язано з тим, що при різних патернах взаємодії та різних видах мап ті чи інші властивості векторних чи растрових плиток можуть стати в нагоді.

2.3 Вибір клієнтської бібліотеки

Після початкового відсіювання, у нас залишилось 3 кандидати. Mapbox, Leaflet та Google Maps. Якщо подивитись офіційний прайсинг Google Maps, то стане зрозуміло, що це рішення явно за рамками наших бюджетів.

Залишається 2 кандидати Leaflet та Mapbox. Обидва базуються на даних з open street map, та є opensource рішеннями, що позитивно впливає на наш бюджет.

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 використовуючи графічне ядро, але вимагає

підтримки openGL для цього. Для сучасних браузерів це не є проблемою [6]. Варто відзначити, 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 [7]. 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.

2.4 Кластеризація та фільтрація

Одночасне відображення 100 000 точок носить досить сумнівну користь, тому що такі об’єми даних у будь-якому випадку будуть займати сотні кілобайт або навіть мегабайти і передавати їх по мережі накладно. Особливо враховуючи, що наша мапа має підтримувати фільтри, а це означає, що при зміні фільтрів потрібно буде знов завантажувати ці об’єкти.

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. При цьому можливо або просто відсікати частину точок і збільшувати їх кількість при збільшенні наближення на мапі, або об’єднувати згруповані точки у кластерний елемент і виводити кількість точок, що були згруповані. Варто також

зазначити, що є два підходи до кластеризації: клієнтська(в браузері) і серверна.

Рисунок 2.4.1 - Кластеризація.

Перевага кластеризації на клієнті - це її простота реалізації і гнучкість, проте вона не зменшує кількість даних, які передаються на клієнт. Крім того, сам процес кластеризації вимагає ресурсів. Тому якщо користувач буде переглядати мапу на low end пристрої, або в роумінгу чи просто в зоні нестабільного інтернету, то він буде мати не найкращі враження. Враховуючи нашу максимально можливу кількість об’єктів - кластеризація на стороні серверу буде більш надійним рішенням.

Фільтрація є однією з наших функціональних вимог. Тут варто зазначити, що поєднання фільтрації при серверній кластеризації накладає деякі обмеження. Справа у тому, що ми не можемо просто згенерувати кластери разом з нашими плитками,

тому що генерація плиток займає суттєвий час, а наперед згенерувати кластери не вдасться, тому що кількість об’єктів у кластері буде різна для різних наборів фільтрів. Якщо генерувати наперед окремі плитки для кожної комбінації фільтрів, то навіть якщо у нас 2 фільра по 5 опцій у кожному - це дасть 25 наборів плиток і кожен наступний фільтр буде збільшувати цю кількість у геометричній прогресії.

Є кілька підходів, щоб вирішити цю проблему: не показувати кількість об’єктів в кластері, генерувати кластери “нальоту”. Перший підхід звичайно простіший. При створенні плиток для мапи у місцях з великим накопиченням точок, деякі точки просто відсікаються на певних рівнях наближення. Чим ближче користувач наближається - тим більше точок він бачить. При цьому більшість користувачів навіть не зрозуміє, що точок було менше, тому що вони і так перекривали одна одну на маленькому збільшенні мапи. Фільтрація у такому випадку забезпечується за рахунок стилізації мапи. Окремий сервіс фільтрації надає інформацію, який набір об’єктів має бути видимий, а mapbox за рахунок стилізації змінює їх зовнішній вигляд, щоб користувачу було зрозуміло, що він відфільтрував. Тут є важливий нюанс у тому, що на маленькому збільшенні не всі об’єкти передаються з сервера плиток, тому що ми їх кластеризували і при певних комбінаціях фільтрів деякі зони 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 в реалізації, але дає кращий досвід нашим користувачам. Для цього ми будемо робити мапу з декількох шарів. Перший шар - це плитки з основною інформацією яку ми бачимо на мапі: вулиці, дороги, будинки. 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 [8].

Кластери потім можуть бути передані як geojson, або конвертовані у векторні плитки. Geojson - більш простий спосіб, але як вже згадувалось раніше - векторні плитки використвують бінарний протокол і мають менший розмір, проте генерація векторних плиток займає більше часу. Тому для досягнення найкращих показників потрібно експериментувати з реальною аудиторією. Тим не менш у більшості випадків прийнятними будуть обидва варіанти, так як на етапі кластеризації буде відкинута основна маса об’єктів. Mapbox підтримує обидва формата, що дає нам достатню маневреність.

2.5 Геокодування та автодоповнення.

Геокодування - це процес знаходження локації об’єкту інтересу на за його назвою. Варто зауважити, що цей сервіс за нашими вимогами має підтримувати автодоповнення. При аналізі поставленої задачі ми вже згадували, що автодоповнення є досить складною задачею. Сервіс має відповідати швидко і витримувати велику кількість конкурентних запитів. На даний момент Mapbox пропонує безкоштовний доступ до їх сервісу геокодування, якщо ви

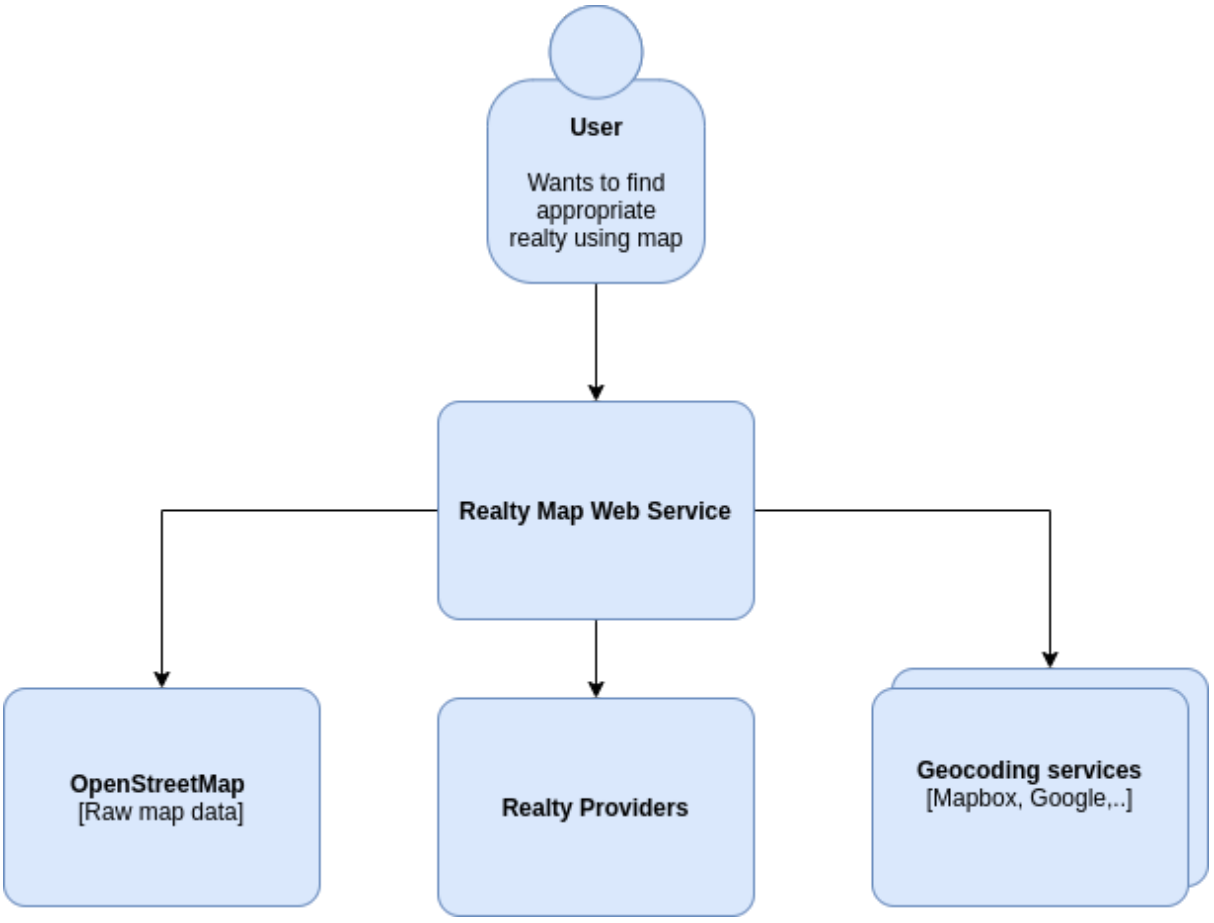
робите до 100 000 запитів на місяць і 0.75$ за кожну наступну тисячу. Після 500 000 ціна за 1 тисячу зменшується до 0.6$.

Варто також зазначити, що API геокодування mapbox нічого не знає про наші об’єкти. Наприклад якщо користувач буде шукати квартиру у новобудові “Чудове житло” - скоріше за все її ще не буде в базі openstreetmap і відповідно у mapbox. Тому для якісної реалізації геокодування нам потрібно буде доповнювати результати своїми об’єктами, або реалізовувати власний сервіс геокодування. При великих навантаженнях власний сервіс буде більш вигідним варіантом у довгостроковій перспективі, враховуючи потенційні ліміти використання, які ми зазначили у нефункціональних вимогах.

Розділ 3. Архітектура.

3.1 Контекстна діаграма.

Ми розглянули основні вимоги до нашої мапи а також технічні особливості, обмеження та варіанти реалізації. Тепер розглянемо систему в цілому.



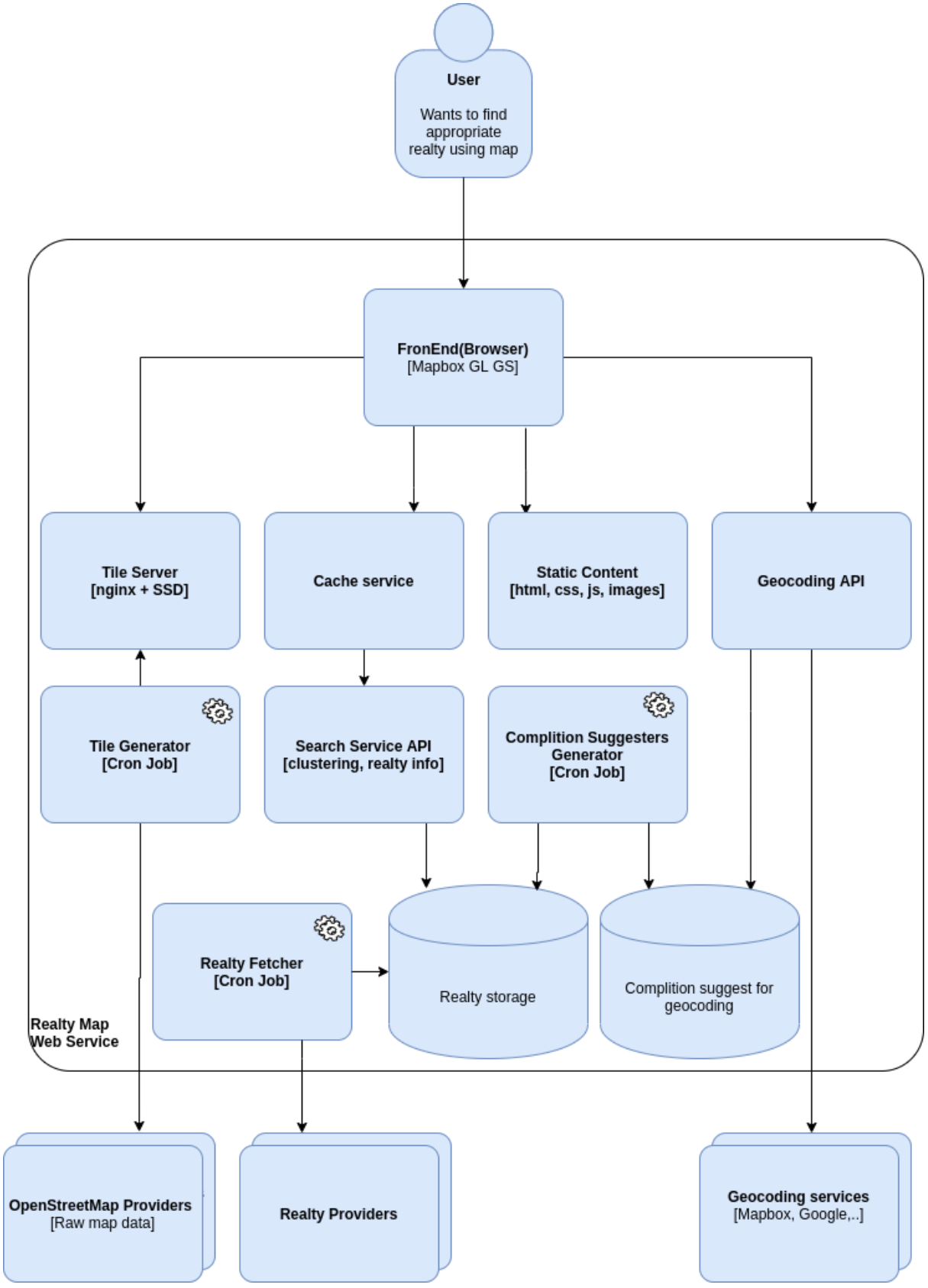
Діаграма 3.1.1 - Контекстна діаграма веб сервісу.

З контекстної діаграми видно, що наш веб сервіс обслуговує користувача, який зацікавлений у пошуку об’єкта нерухомості на мапі. Наш сервіс отримує сирі дані з OpenStreetMap, доповнює їх даними про розташування об’єктів нерухомості, та

використовує geocoding від різних провайдерів, для реалізації функціоналу автодоповнення.

Є кілька сервісів, які надають можливість безкоштовно завантажити актуальні дані OpenStreetMap: Overpass API, Planet OSM, Geofabrik Downloads,.. та інші. На даному етапі проектування ми не будемо обмежуватись конкретним сервісом. Дані про об’єкти нерухомості можна завантажити використовуючи відкритий API одного з найбільших порталів нерухомості України - DOM.RIA. Сервісів геокодування також є достатня кількість на ринку і ми не будемо обмежуватись на даному етапі, так як не знаємо реальні патерни взаємодії користувача і, в залежності від об’ємів використання, можемо повністю від них відмовитись у випадку виходу за ліміти бюджету.

3.2 Компонентна діаграма



Розглянемо структуру нашого веб сервісу більш детально на компонентній діаграмі.

Користувач взаємодіє з системою через наш фронтенд(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. Також при подальшому розвитку і необхідності сервер статики може доставляти один із сучасних фронтенд фреймворків, наприклад React, для того щоб реалізовувати більш складні патерни взаємодії на клієнтській стороні. Сервер, який займається роздачею статики, може бути один, або кілька серверів, об’єднані в CDN мережу, в залежності від навантаження, вимог до швидкодії та доступності. Amazon надає змогу безкоштовно використовувати свій CDN під назвою CloudFront, при лімітах 50 ГБ і 2 млн. запитів у місяць, що може бути хорошим варіантом для старту.

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 брати сирі дані з OpenStreetMap генерувати плитки та складати їх на сервіс, який буде займатись їх роздачею.

Mapnik opensource продукт, який вміє генерувати як растрові, так і векторні плитки, має хорошу інтеграцію з іншими інструментами та підтримку найпопулярніших мов програмування в цій області: Nodejs, Python. За замовчуванням mapnik не вміє генерувати векторні плитки, але є додатковий модуль, який додає таку можливість.

Крім основної частини мапи, на ній мають бути ще об’єкти нерухомості, які й мають створювати додаткову цінність для нашого користувача. За їх доставку, фільтрацію та надання додаткової інформації буде відповідати Search Service. Він може передавати кластеризовані результати пошуку у вигляді GeoJson або векторних плиток. Дані він буде брати з бази Realty Storage. Це може бути практично будь-яка база даних, так як у нас немає жорстких вимог до структури, транзакційності чи доступності, а 100 000 об’єктів це помірний об’єм для будь-якої сучасної БД. Так як дані будуть змінюватись не дуже часто, то при виборі реляційної бази з індексами по пошуковим полям можна добитись досить високої швидкодії. PostgreSql, крім того, надає можливість реалізувати повнотекстовий пошук, що може бути корисною додатковою функціональністю.

Операції пошуку по БД хоч і оптимізуються за рахунок індексів, але все одно відносно ресурсоємні і можуть стати вузьким місцем при збільшенні навантаження. Те ж саме можна сказати про обчислення кластерів та генерацію векторних плиток на основі GeoJson. Для того щоб забезпечити максимальну продуктивність та стабільність системи, ми додали сервіс кешування результатів обчислення. Таким чином більшість запитів буде оброблятись за десяті долі секунди і віддаватись з кешуючого сервісу.

Realty Fetcher - сервіс який буде запускатись по необхідності для того щоб синхронізувати базу даних провайдера об’єктів нерухомості і нашу базу. Він потрібен для того, щоб не створювати надмірне навантаження на API провайдера об’єктів нерухомості. Також це покращить доступність нашої системи, так як наша

система буде працездатна, навіть якщо провайдер якийсь час буде недоступний. Крім того окрема база робить нас незалежними від функціональності API провайдера. Наприклад, якщо провайдер не надає можливості пошуку об’єктів по кімнатності, то ми можемо реалізувати це самі, при наявності даних. Крім того, ми можемо таким чином інтегруватись з різними провайдерами даних і виступати агрегатором. На ранніх етапах розробки можливо напряму звертатися до API провайдера і додати базу синхронізації по необхідності.

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. 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. За рахунок того, що база підказок зберігається в оперативній пам’яті - гарантує високу швидкодію. Є також готові рішення від cloud провайдерів.

Таким чином ми розглянули основні компоненти системи, та їх конкретні варіанти реалізації. Дана архітектура не залежить від конкретного cloud провайдера, також може бути self hosted

рішенням. Також архітектура легко масштабується під будь-яке навантаження і бізнес потреби. 100 000 об’єктів - це швидше початок ніж межа. За рахунок серверної кластеризації система може масштабуватись в десятки разів. Nginx - де-факто стандарт в індустрії для роздачі статичного контенту і може обслуговувати тисячі користувачів одночасно. Також варто зазначити, що ми не залежимо від жодного зовнішнього провайдера, так як їх завжди є декілька.

Розділ 4. Висновок.

Сучасні підходи в проектуванні архітектури сильно відрізняються від тих, що були 10 - 20 років тому. Це пов’язано з динамічністю розвитку індустрії і бізнесу. Agile методологія у багатьох сферах витісняє Waterfall. І проектування архітектури не вийняток. Якщо раніше хороша архітектура - це однозначне рішення, яке враховує всі нюанси і не дозволяє зробити крок в бік, то зараз хороша архітектура - це та, яка може швидко адаптуватись, реалізовуватись та націлена в першу чергу на задоволення бізнес потреб.

Саме тому проектування, в першу чергу, починається з виявлення основних функціональних та нефункціональних вимог і в результаті має показати, яким чином саме ця архітектура надає можливість задовольнити ці вимоги.

Можливість швидко реалізувати першу робочу версію системи також дуже важливий фактор. Для прикладу, дану систему можна реалізувати уже в першій ітерації, якщо не робити власний сервер плиток, а використовувати безкоштовний ліміт сервісу mapbox. Також можна для початку обмежитись одним провайдером нерухомості і відмовитись від синхронізуючої бази. Геокодування може на першій ітерації не враховувати підказки з наших даних. Таким чином ми отримаємо робочу версію системи вже на першій ітерації. Отримаємо фідбек від наших стейкхолдерів, реальні дані швидкодії і будемо краще розуміти в якому напрямку слід розвивати систему.

Витрати також дуже важливий фактор. Бажано, щоб витрати були мінімальними на перших етапах - це дозволить бізнесу з мінімальними ризиками випробувати нову систему. Ми можемо запустити MVP при мінімальних витратах на безкоштовних лімітах провайдерів послуг. Також в хорошій системі витрати мають зростати повільніше, ніж навантаження на неї. В нашій архітектурі це реалізовано за рахунок використання opensource рішень, попередньої генерації даних, кешування, можливості зміни cloud провайдера, відсутності ресурсоємних компонентів.

Хороше архітектурне рішення має завжди враховувати потреби бізнесу до росту. Система, яка витримує навантаження сьогодні, але буде непрацездатна через 2 роки, через неможливість масштабуватись, як правило, мало корисна.

Також важливо звернути увагу на якість діаграм. Хороша діаграма має легко читатись, а кожен її елемент - слугувати для пояснення стейкхолдерам, яким саме чином будуть задоволені їх вимоги до системи.

“If change is the only constant in the universe, then software change is not only constant but ubiquitous.”(Len Bass, Software Architecture in Practice)

Список використаної літератури

1. Eugene Stepnov. TOP JAVASCRIPT MAPS API AND LIBRARIES. 2016 [Електронний ресурс] - Режим доступу: https://flatlogic.com/blog/top-javascript-maps-api-and-libraries/
2. Sample, J.T. Tile-Based Geospatial Information Systems. Principles and Practices; Springer: Berlin/Heidelberg, Germany, 2010.
3. Stefanakis, E. Map Tiles and Cached Map Services. GoGeomatics. Magazine of GoGeomatics Canada. 2015. [Електронний ресурс] - Режим доступу: http://www2.unb.ca/~{}estef/papers/go_geomatics_stefanakis_december_2015.pdf
4. Noskov, A. Computer Vision Approaches for Big Geo-Spatial Data: Quality Assessment of Raster Tiled Web Maps for Smart City Solutions. In 7th International Conference on Cartography and GIS; Bulgarian Cartographic Association: Sofia, Bulgaria, 2018; pp. 296–305. [Електронний ресурс] - Режим доступу: http://dx.doi.org/10.5281/zenodo.1346671
5. Antoniou, V.; Morley, J.; Haklay, M.M. Tiled Vectors: A Method for Vector Transmission over the Web. In Proceedings of the International Symposium on Web and Wireless Geographical Information Systems, Maynooth, Ireland, 7–8 December 2009; Springer: Berlin/Heidelberg, Germany, 2009; pp. 56–57. [Електронний ресурс] - Режим доступу: http://link.springer.com/10.1007/978-3-642-10601-9_5

6. Mapbox. Browser support. [Електронний ресурс] - Режим доступу: https://docs.mapbox.com/help/troubleshooting/mapbox-browser-support/
7. Mapbox. Leaflet. [Електронний ресурс] - Режим доступу: https://docs.mapbox.com/help/glossary/leaflet/
8. Paul Nebel. Dynamic Server-Side Clustering for Large Datasets. 2018. [Електронний ресурс] - Режим доступу: https://geovation.github.io/dynamic-server-side-geo-clustering