МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики
(повне найменування назва факультету)

Кафедра інформаційних систем
(повна назва кафедри)

Магістерська робота

на тему:

Розробка вебсервера
на основі WebSocket протоколу

Виконав: студент групи ПМіМ-22с
спеціальності
122 «Комп'ютерні науки»
(шифр і назва спеціальності)

Кізло Т. М.
(підпис) (прізвище та ініціали)

Керівник доц. Бернакевич І. Є.
(підпис) (прізвище та ініціали)

Рецензент Коковська Я.В.
(підпис) (прізвище та ініціали)

Львів – 2021

ЗМІСТ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет *прикладної математики та інформатики*
Кафедра *інформаційних систем*
Освітньо-кваліфікаційний рівень *магістр*
Напрям підготовки
Спеціальність 122 «Комп'ютерні науки»

«ЗАТВЕРДЖУЮ»

Зав. кафедрою *проф.Шинкаренко Г.А.*

« 16 » *вересня* 2020 р.

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТА

Кізла Тараса Михайловича
(прізвище, ім'я, по батькові)

1. Тема роботи
Розробка вебсервера на основі WebSocket протоколу

керівник роботи *доц. Бернакевич І.Є.*
затверджені Вченою радою факультету від «_» _____ 20_р., №
2. Строк подання студентом роботи
3. Вихідні дані до роботи
Література та інтернет-ресурси за тематикою роботи

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд сучасного стану проблеми
2. Постановка задачі
3. Розробка архітектури аплікації
4. Програмна реалізація
5. Апробація
6. Оформлення роботи

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Презентація магістерської роботи
Знімки екрану вебзастосунку
Діаграма варіантів використання
Огляд архітектури системи
Діаграма розгортання аплікації
Діаграми послідовностей взаємодії сервісів
Результати вимірів продуктивності
Результати вимірів стійкості до навантаження
Схема моделі загроз

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№	Найменування етапів дипломної (кваліфікаційної) роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд сучасного стану проблеми*	*вересень 2020*	
2.	*Постановка задач*	*жовтень 2020*	
3.	*Розробка архітектури аплікації*	*листопад 2020 – грудень 2020*	
4.	*Програмна реалізація*	*січень 2021 - червень 2021*	
5.	*Апробація*	*липень 2021 – жовтень 2021*	
6.	*Оформлення роботи*	*листопад 2021*	

Студент _____ *Кізло Т. М.*
підпис

Керівник роботи _____ *доц Бернакевич І.Є.*
підпис

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Асинхронний JavaScript і XML (англ. Asynchronous JavaScript And XML, AJAX) — підхід до побудови користувацьких інтерфейсів вебзастосунків, за яких вебсторінка, не перезавантажуючись, у фоновому режимі надсилає запити на сервер і сама звідти довантажує потрібні користувачу дані.

Атака на відмову в обслуговуванні (англ. DoS attack) — тип загрози вебаплікацій, який полягає в намірі зробити комп'ютерні ресурси недоступними, шляхом збільшення навантаження на сервер.

Багатосторінковий застосунок (англ. Multi-Page Application, MPA) — вебсайт, який вміщується на багатьох сторінках.

База даних, БД (англ. Database) — впорядкований набір даних.

Веббраузер, браузер (англ. Browser) — програмне забезпечення, що дає можливість користувачеві взаємодіяти з вебсервером.

Вебклієнт, Клієнт (англ. Web client) — апаратний або програмний компонент обчислювальної системи, який надсилає запити серверу.

Вебресурс (англ. Web resource) — це інформаційні ресурс, який призначені для забезпечення освіти, розміщені у вебпросторі у вигляді різних форматів (текстового, графічного, архівного, аудіо- та відеоформатів і т.д.).

Вебсайт або сайт (англ. Website) — сукупність вебсторінок та залежного вмісту, доступних у мережі Інтернет, які об'єднані як за змістом, так і за навігацією під єдиним доменним ім'ям.

Вебсервер (англ. Web server) — це сервер, що приймає запити від клієнтів, зазвичай веббраузерів, видає їм відповідь.

Вебсокет (англ. WebSocket) — це протокол, що призначений для обміну інформацією між браузером та вебсервером в режимі реального часу.

Вебсторінка (англ. Webpage) — інформаційний ресурс, доступний в мережі, який можна переглянути у веббраузері.

Джерело подій (англ. Event sourcing) — це підхід в проєктуванні, при якому стан об'єкта представляється у вигляді множини подій.

Ідемпотентність (англ. Idempotence) — це властивість при якій повторна дія над будь-яким об'єктом не змінює його результату.

Неперервна інтеграція/Неперервне розгортання (англ. Continuous Integration/Continuous deployment, CI/CD) — набір практик розробки програмного забезпечення, направлених на автоматизацію інтеграційних проблем.

Односторінковий застосунок (англ. Single-Page Application, SPA) — вебсайт, який вміщується на одній сторінці з метою забезпечити користувачу досвід близький до користування настільною програмою.

Предметно-орієнтоване проєктування (англ. Domain-Driven Design, DDD) — це підхід до моделювання складного програмного забезпечення.

Прикладний програмний інтерфейс (англ. Application Programming Interface, API) — набір методів для взаємодії із програмним забезпеченням.

Протокол передачі гіпертекстових документів (англ. HyperText Transfer Protocol, HTTP) — протокол передачі даних, що використовується в комп'ютерних мережах.

Розділення відповідальності між командами та запитами (англ. Command-Query Responsibility Segregation, CQRS) — принцип в програмуванні, що вказує, на те що метод повинен бути або командою, яка виконує якусь дію, або запитом, що повертає дані, але не одночасно і тим, і іншим.

Сервер (англ. Server) — комп’ютер здатний реагувати на зовнішні події відповідно до встановленого програмного забезпечення.

ВСТУП

На сьогодні важко уявити наше життя без інтернету. За декілька десятків років він перетворився з абсолютно чужого та незвіданого поняття до так всім звичного та повсякденного явища.

Глобальна мережа набула широкої популярності серед мільйонів людей. Зараз багато хто з нас не уявляє життя без цього геніального винаходу. Хоча на початку свого розвитку він був лише розкішшю обмеженого кола людей.

Та навіть коли Інтернет став доступний широкій масі, він виглядав далеким від того до чого ми звикли. Його розвиток обумовлений не лише неосяжними вимогами людей, які проявлялись в бажані отримувати більше інформації за менший проміжок часу, але й також різкому розвитку технологій, що припав на наш період.

Таким чином, Інтернет, за період свого розвитку пройшов декілька етапів еволюції. Важко виділити межі одної чи іншої фази, та все ж таки, ми можемо побачити основні ознаки притаманні кожному періоду розвитку.

Так при своїй появі, користувачі могли лише насолоджуватись незмінним контентом, що був переповнений текстом. В цей момент Всесвітня павутина не відрізняється сильно від онлайн-довідників чи енциклопедії. Різноманіттям можуть похвалитись лише сайти-візитки та каталоги.

Вимоги ненаситних користувачів швидко зростали, тому своєю появою не забарився новий етап розвитку Інтернету. Вміст інтернет-сторінок більше не є суцільним текстом. Появляються різноманітні графічні елементи на зразок картинок, кольорових панелей тощо. Паралельно розвивається технологія Ajax, яка дозволяє уникнути повторного оновлення вебсторінки, для отримання інформації. Вебсайти стають більше орієнтованими на повсякденного користувача. Додається інтерактивність, та спрощується сам контент. Окрім цього, розвивається можливість взаємодії з іншими користувачами. Ера соціальних мереж розпочинає своє панування.

Наразі Інтернет зміг перескочити планку розвитку втретє. Ми без труднощів використовуємо високо якісний вебконтент, який націлений на задоволення потреб користувачів. Інтернет став засобом комунікації, що обслуговує базові потреби людей по всьому світу. Він не лише основне джерело інформації та знань, але й спосіб уникнути реальності та з головою пірнути у світ пригод і розваг.

Кількість користувачів Інтернету є надзвичайно великою і продовжує невпинно зростати. Як бачимо, його розвиток, був обумовлений рядом тенденцій, які стали визначальними. Їх підґрунтям є нові технології та підходи до розробки, підтримки, використання вебресурсів, обміну інформацією між ними тощо.

Якщо проаналізувати розвиток послуг Глобальної мережі, то можна побачити яким чином збільшувались вимоги користувачів. Ми перейшли від сторінок наповнених статичним контентом, до вебсайтів, які зачаровують своїм безмежним різноманіттям та неосяжними можливостями. Ми більше не обмежені у використанні лише текстових матеріалів. У нас появилась можливість у застосуванні зображень, малюнків, аудіо- та відеофайлів тощо. Нам більше не доводиться очікувати перезавантаження усієї сторінку з метою отримання нової інформації. Вебсайти стали подібними до звичайних програм і тепер лише необхідна секція оновлює свій вміст. Здається ніби весь Інтернет націлений на зручне використання кінцевим користувачем та більше немає куди рухатись далі. Саме всупереч останньому тезису і виникла ідея розробки нового покоління вебсерверів.

МЕТА

Суть даного завдання є створення вебсерверу на основі WebSocket протоколу. Дана тема обрана не випадково. Як зазначалось раніше Інтернет перебуває на піку свого розвитку, та сучасні вебсайти все одно не використовують усього можливого потенціалу.

Більшість вебресурсів націлені на взаємодію між користувачами. Так різноманітні аплікації для спілкування, відомі також, як вебчати, пройшли етап, від використання HTTP протоколу, коли оновлення контенту відбувалось із певною заданою періодичністю, до використання WebSocket протоколу, коли повідомлення передаються безпосередньо від одного співрозмовника до іншого.

Наразі вебсторінки, хоч і відрізняються від своїх “предків”, та все одно не можуть похвалитись таким самим високим рівнем інтерактивності, як чати.

Метою цієї роботи є використання WebSocket протоколу для побудови інтерактивних вебсторінок, дослідження складності розробки, затратності ресурсів та власне інтерактивні можливості отриманого застосунку.

Дана задача не передбачає використання у якійсь конкретній предметній області, а поширюється на великий спектр застосування. Таким чином, будь-який застосунок, будь то блог, соціальна мережа, хмарне сховище тощо, зможе отримати переваги застосувавши даний підхід. Варто зазначити, що все одно існують галузі, для яких даний підхід є занадто складним, наприклад, сайти-візитки, довідники тощо. Така модель чудово проявляється лише у системах з високим рівнем динамічності контенту.

Незважаючи про те, що предметна область довільна, вона не може залишатись невизначеною, оскільки без неї ми не зможемо вирішити поставленого завдання. Нехай предметною областю буде система управління установами для транспортування вантажів.

Така система буде корисна для більшості поштових організацій, оскільки вона забезпечить уніфікований сервіс для поштових компаній, де вони можуть

керувати замовленнями, а клієнти своєю чергою зможуть перевірити статус замовлення, *рисунок 1.1*:

Рисунок 1.1 — Огляд предметної області.

Задана предметна область може видатись напрочуд простою, але вона і не повинна бути складною. Її основна мета — це слугувати прикладом для розв’язування основної задачі, а не захмарювати її своєю складністю.

Окрім цього можна побачити, що задана модель чудово підходить для розв'язання поставленої задачі, оскільки взаємодія між клієнтом і працівником побудована на використанні механізму подій. Так, якщо працівник змінить інформацію про замовлення, його статус чи місцеперебування, клієнт повинний дізнатись про це без зайвого перезавантажування сторінки. Описаний характер і ступінь взаємодії між об'єктами не є штучним, а вимушений заданою предметною областю.

Оскільки для багатьох предметних областей важливим фактором є не лише продуктивність взаємодії, а також ступінь захищеності, дане питання теж буде розглянуте в ході виконання даної роботи.

1 ПОСТАНОВКА ЗАДАЧІ

1.1 Фізична постановка задачі

Існує два типи користувачів, які взаємодіють із системою: працівники, так і звичайні клієнти поштових закладів. Кожен із цих двох типів користувачів по-різному використовує систему, тому кожен із них має свої вимоги.

Клієнти можуть використовувати програму лише для перегляду хронології замовлення, історії замовлень та отримання SMS-сповіщень. Адміністратори керуватимуть замовленнями, наприклад, створюватимуть замовлення, додаватимуть та вилучатимуть вантажі, редагуватимуть місце поточного замовлення, видалятимуть замовлення тощо. Вони керують, як і загальною системою, так і інформацією для кожного замовлення.

Основний набір дій, можна представити у вигляді варіантів використання, *рисунок 1.1*.

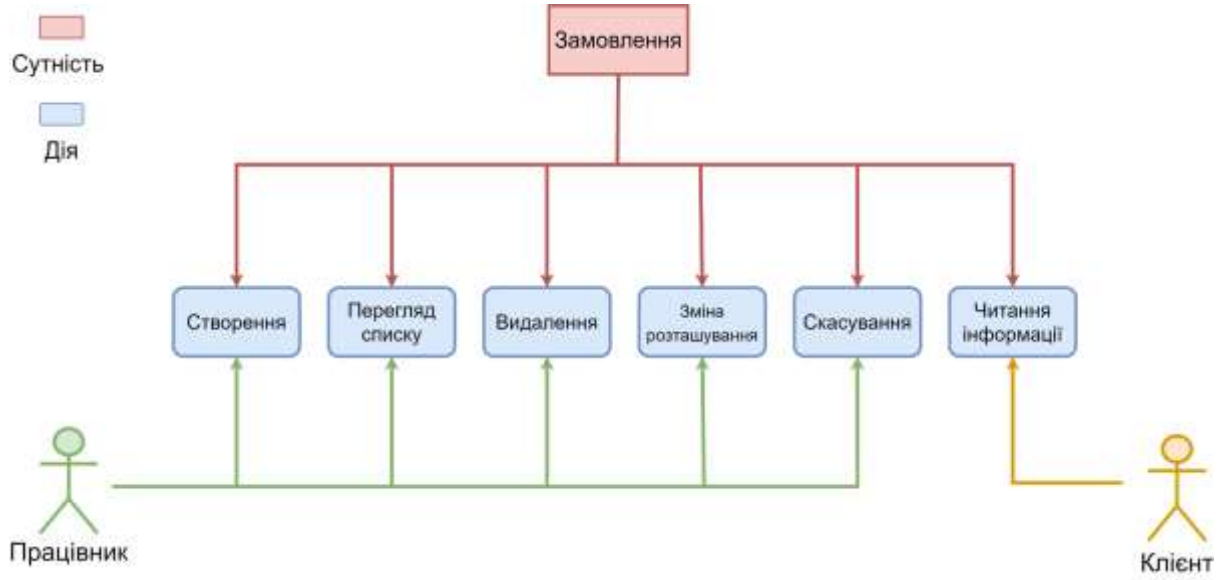

Рисунок 1.1 — Діаграма варіантів використання.

Різновид використання описує, «хто» і «що» може зробити з розглянутою системою, тобто графічно описує поведінку системи, як вона відповідає на зовнішні запити.

Вимоги до програмного забезпечення, які описують внутрішню роботу системи, її поведінку та інші специфічні функції, які має виконувати система, називають також функціональними вимогами.

Розглянемо список вимог зі сторони працівника поштового відділення:

а) працівник повинен бачити панель «Зареєструвати замовлення»;
б) співробітник повинний вказати опис замовлення, телефон та місцеперебування як відправника, так і одержувача;
в) замовлення повинне містити як мінімум один вантаж;
г) вантаж повинен мати інформацію про ширину, висоту та довжину;
д) працівник повинен мати можливість додавати та вилучати вантаж під час створення замовлення;
е) процес створення замовлення повинний завершуватись підтвердженням зі сторони співробітника;
ж) співробітник повинен мати змогу переглянути список замовлень за допомогою панелі «Перегляд списку замовлень»;
з) працівник повинен бачити дійсний номер ТТН для кожного замовлення;
и) працівник бачить статус кожного замовлення (Нове замовлення, В процесі доставлення, Доставлено, Виконано, Скасовано);
к) працівник повинен мати можливість видалити замовлення із системи;
л) працівник має можливість скасувати замовлення;
м) працівник повинен мати змогу змінити поточне місце розташування замовлення (місто та вулиця);
н) статус замовлення повинний автоматично змінюватись, коли працівник змінює поточне місцеперебування замовлення, скасовує замовлення тощо;

Перелік описів функціональних потреб клієнтів поштового відділення:

а) клієнт після створення замовлення повинен отримати номер ТТН у Смс-повідомленні;
б) клієнт повинен мати доступ до панелі «Переглянути історію замовлення»;
в) клієнт повинен мати змогу переглянути графік замовлення за номером ТТН, отриманим зі Смс;

г) клієнт повинен мати змогу завершити замовлення, якщо воно йому вже доставлено;

1.2 Програмна постановка задачі

Як можна побачити дана програма передбачає велику кількість можливостей для користувачів. Та окрім функціональних вимог, є набір критерій до програмного забезпечення, які задають характеристику для оцінки якості його роботи. Вони визначають якою система повинна бути. Так нижче наведено список нефункціональних вимог, щодо продуктивності, надійності, безпеки тощо.

Перелік вимог до продуктивності:

а) додаток повинний завантажуватись протягом 2 секунд;

б) система повинна мати можливість підтримувати 1000 одночасних користувачів;

в) час відповіді на запити та оновлення не повинен перевищувати 500 мс;

Перелік вимог надійності:

а) система повинна працювати, навіть якщо сталися помилки;

б) система повинна бути доступною 75% часу протягом місяця;

в) час простою після критичної несправності не повинен перевищувати 5 годин. Середній час відновлення повинен становити 2-5 годин;

Перелік вимог щодо зручності використання:

а) користувач бачить усі зміни своїх або інших користувачів динамічно, без перезавантаження сторінки із затримкою не більше 250 мс;

б) система повинна працювати як вбудований додаток без перезавантаження сторінки;

в) інтерфейс повинен бути легким для вивчення без підручника та дозволяти користувачам без помилок досягти своїх цілей;

Перелік вимог до підтримки програмного забезпечення:

а) додаток повинен використовувати безперервну інтеграцію, щоб функції та виправлення помилок могли бути швидко розгорнуті без простою;

б) система повинна бути простою для тестування;

Перелік вимог щодо безпеки:

а) дозвіл на доступ до програми може змінювати лише адміністратор системи;

б) вимоги до пароля - довжина, спеціальні символи, термін дії;

в) усі зовнішні зв’язки між сервером та клієнтами повинні бути зашифровані;

2 АРХІТЕКТУРА

Як можна побачити дана програма передбачає велику кількість можливостей для користувачів. Тепер коли ми визначились, які вимоги накладаються на нашу програму, залишилось вирішити за допомогою якої мови програмування буде побудована ця інформаційна система. Також варто вибрати технології розробки та платформу на якій буде розгортатись програма.

Щоб спростити процес розробки та уникнути багатьох типових помилок варто виділити час на проєктування та розробку архітектури. Спробуємо як найкраще розбити систему на частини, описати їх взаємодію одна з одною, те як між ними передається інформація. Розглянемо як ці частини розвиваються поодинці та опишемо все використовуючи формальну чи неформальну нотацію.

Розглянемо такі аспекти розробки, як:

а) архітектура клієнта;

б) створення об'єктноорієнтованої моделі предметної області;

в) архітектура сервера;

г) модель даних;

2.1 Архітектура клієнта

Основними підходами при розробці вебаплікації являються односторінкові та багатосторінкові застосунки. Розглянемо їх переваги та недоліки, *згідно з таблицею 2.1*:

Таблиця 2.1 — Порівняння односторінкових та багатосторінкових застосунків

	Односторінковий застосунок (SPA)	Багатосторінковий застосунок (MPA)
переваги	– кращий досвід у використанні для користувача, оскільки не	– простота;

	відбувається перезавантаження сторінки; – швидкодія; – незалежність розробки клієнтської аплікації та сервера;	– Оптимізація для пошукових систем; – надійніший захист;
недоліки	– складність; – навантаження на клієнта;	– оновлення сторінки на кожну дію;

У багатосторінкових застосунках сервер присилає повноцінну сторінку клієнту на кожний запит, *рисунок 2.1*. Що приводить до оновлення сторінки. Даний підхід популярний серед багатьох сайтів, наприклад, Habr чи DOU.

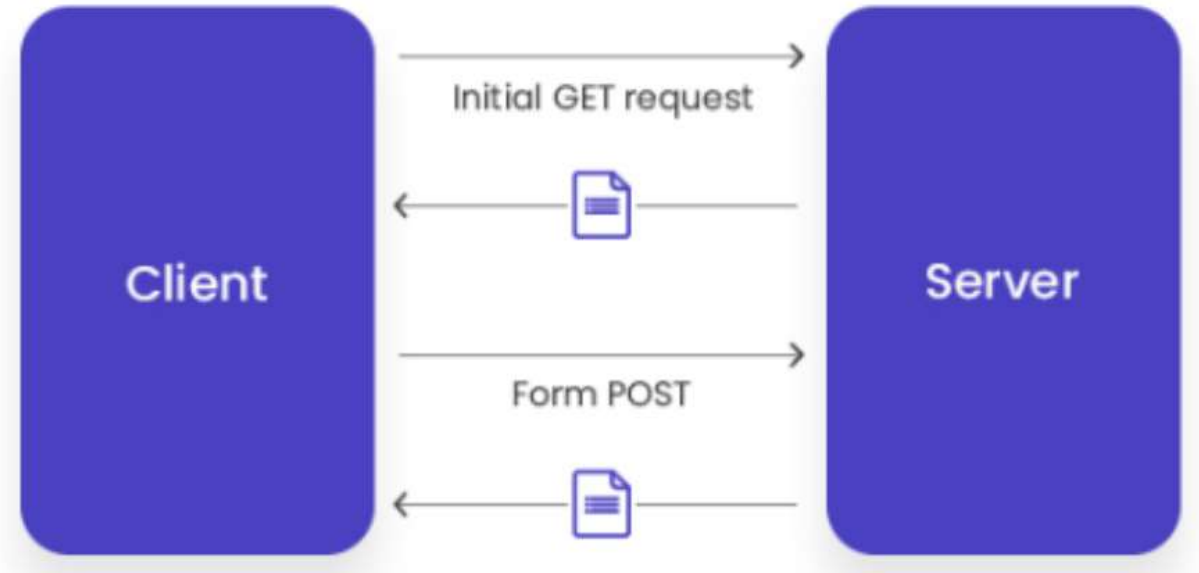

Рисунок 2.1 — Приклад взаємодії між клієнтом та сервером для багатосторінкових застосунків.

В той час як в односторінкових застосунках сервер присилає лише необхідні дані у визначеному наперед форматі, *рисунок 2.2*. Це зменшує розмір передачі даних та призводить до оновлення не всієї сторінки, а лише необхідного компонента. Типові представники Youtube, Google Mail чи Twitter.

Рисунок 2.2 — Приклад взаємодії між клієнтом та сервером для односторінкових застосунків.

Обидва підходи, актуальні на цей момент, та все ж завдяки своїй інтерактивності та кращому досвіді для користувача реалізуємо підхід односторінкових аплікацій.

2.2 Обумовленість використання WebSocket протоколу

Вебаплікація передбачає поділ системи на клієнта (частину із якою взаємодії користувач) та сервер (важко навантажений модуль, що здійснює всі обчислення та відповідає із взаємодію з клієнтами), *рисунок 2.3:*

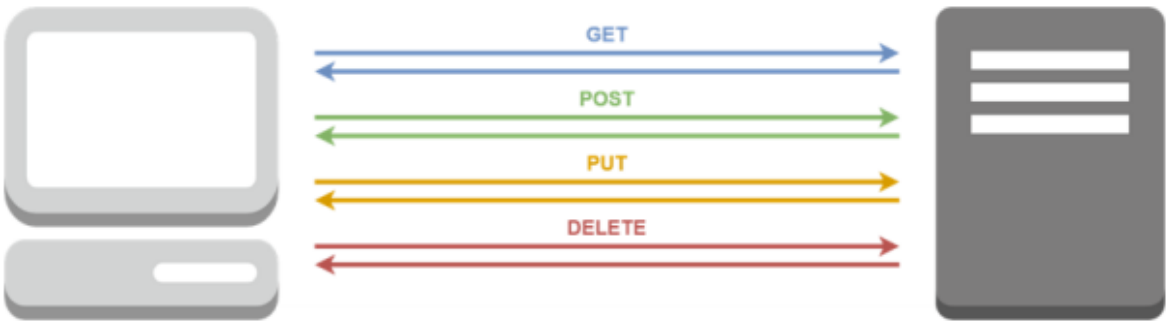

Рисунок 2.3 — Модель взаємодії клієнт-сервер.

При цьому доступ до інформаційних ресурсів зазвичай здійснюється за допомогою методів HTTP запиту, згідно з *таблицею 2.2*. Таким чином кожна із дій користувача відповідає наступним методам:

Таблиця 2.2 — Приклад відповідності дій користувача HTTP методам

Дія	HTTP метод
Читання даних	Get
Створення даних	Post
Редагування даних	Put
Видалення даних	Delete

Даний підхід є типовим рішенням і не дивно, адже він простий в реалізації та звичний у користуванні, *рисунок 2.4*:

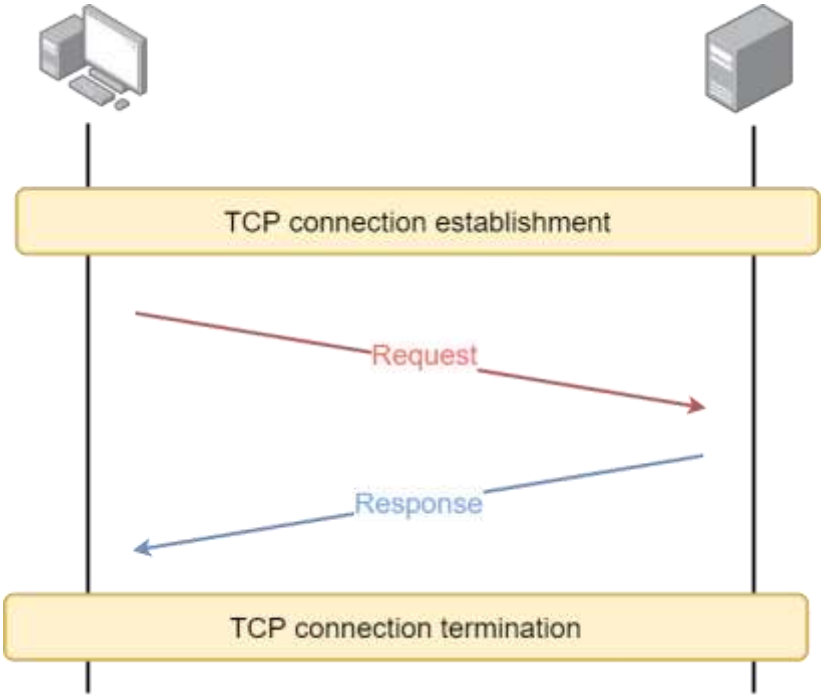

Рисунок 2.4 — Модель роботи протоколу HTTP 1.0.

Та спробуємо розглянути основні недоліки такого підходу. Перш за все модель запит-відповідь вимагає встановлення та розриву з’єднання для отримання кожного окремого ресурсу. Це не лише витрачає ресурси як клієнта, так і сервера, але та ускладнює завантаження великої кількості даних. Також при використанні pull моделі немає можливості дізнатись про зміни від інших користувачів без надсилання повторного запиту. Одним із рішень для розв'язання даної проблеми є

long polling підхід, при якому повторно із деяким інтервалом відправляються запити від клієнта на сервер.

Впродовж багатьох років розроблялись способи покращення HTTP протоколу. Так в наступних версіях був доданий заголовок keep-alive, що дозволяє підтримувати з'єднання між клієнтом і сервером нерозривним впродовж деякого часу, *рисунок 2.5:*

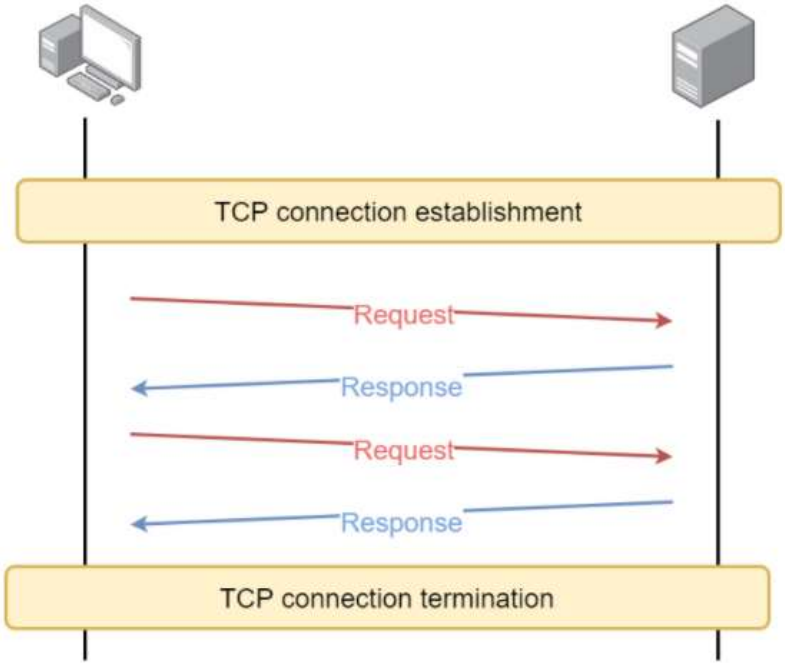

Рисунок 2.5 — Модель роботи протоколу HTTP 1.1 з використанням заголовку keep-alive.

Це зменшило накладні витрати на встановлення з'єднання, але проблеми завантаження великої кількості ресурсів та отримання сповіщень від сервера залишились відкритими.

Протокол HTTP 2.0 приніс ряд наступних покращень

а) Стиснення заголовків. Нова технологія використовує кодування Хаффмана, для стиснення заголовків.

б) Бінарна серіалізація даних. Дані більше не передаються у чистому вигляді, а серіалізуються. Шляхом зменшення розміру пакетів, що передаються, зростає і швидкість передачі.

в) Мультиплексування. Можливість використання одного з’єднання, для обробки декількох запитів, *рисунок 2.6*. При цьому запити працюють паралельно не блокуючи один одного. Як результат, зменшується затримка передачі оскільки використовуються максимальні пропускні можливості мережі.

г) Можливість використання технології push. У попередніх версіях HTTP сервер може зв’язуватися з клієнтом лише тоді, коли до нього надходить запит. Однак тепер сервер може надсилати кілька відповідей на один запит. Під час цього відкритого з’єднання сервер крім відповіді, може передавати клієнту іншу корисну інформацію. Наприклад, під час завантаження вебсторінка сервер може почати відправляння файлів стилів і при цьому проштовхнути клієнту сповіщення від інших користувачів.

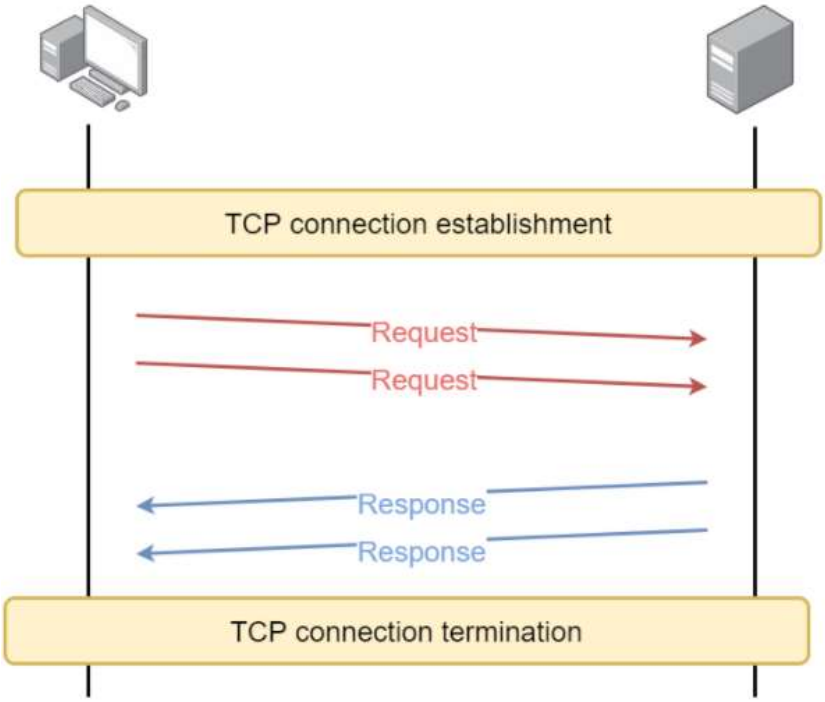

Рисунок 2.6 — Модель роботи протоколу HTTP 2.0 з використанням мультиплексування.

Нова версія не лише покращила продуктивність роботи протоколу, але й відкрила можливості для двосторонньої взаємодії. На практиці можливість зворотної взаємодії не зазнала великої популярності з однієї причин: сервер може проштовхнути клієнту інформацію лише під час відкритого з’єднання. Це означає,

що зворотна передача можлива лише з ініціативи клієнта, а отже оновлення інформації відбувається не в реальному часі, а на вимогу.

В результаті даних обмежень сучасні вебаплікації для взаємодії між клієнтом і сервером використовують два протоколи: HTTP — для отримання даних методом pull та WebSocket взаємодію у реальному часі методом push, *рисунок 2.7:*

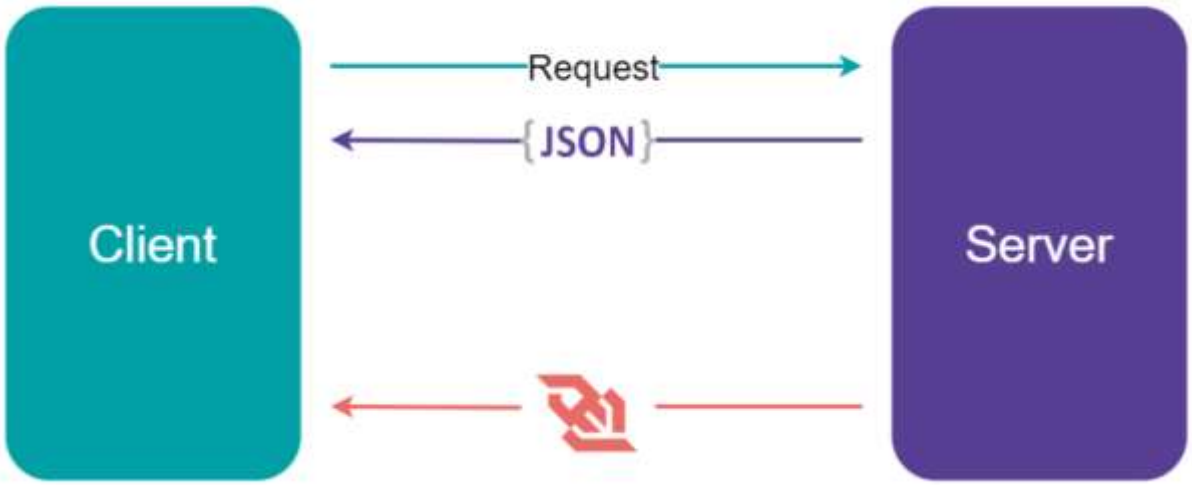

Рисунок 2.7 — Модель взаємодії клієнт-сервер із використанням протоколів HTTP та WebSocket.

При цьому немає причин затрачати ресурси на використання HTTP протоколу. Протокол WebSocket є двостороннім, що дозволяє передавати дані як з клієнта на сервер, так і навпаки, *рисунок 2.8*:

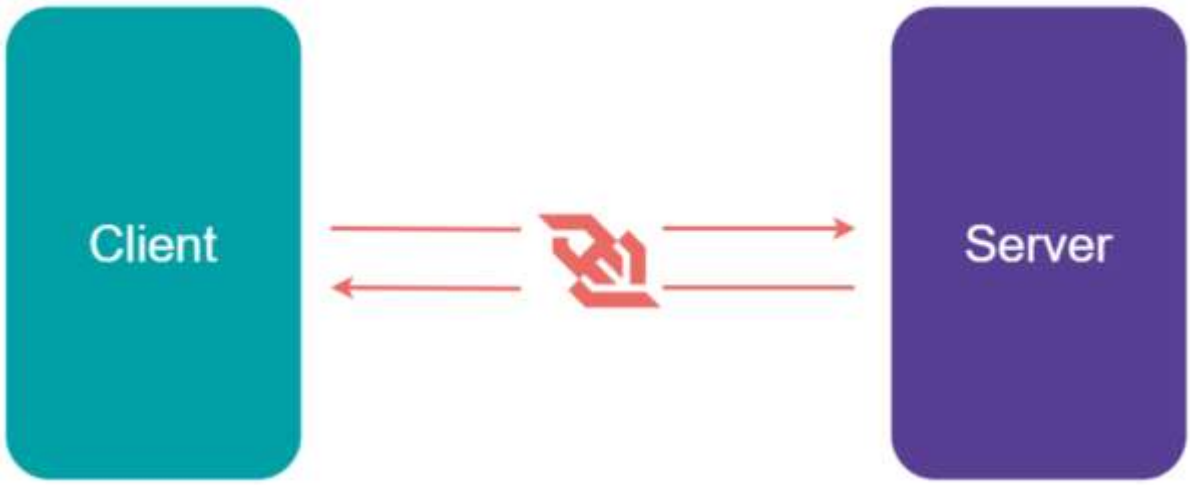

Рисунок 2.8 — Модель взаємодії клієнт-сервер із використанням протоколу WebSocket.

Якщо обмежитись використанням лише цього протоколу ми не лише уникаємо обмежень HTTP протоколу, але й використовуємо уніфікований спосіб взаємодії.

2.3 Архітектура сервера

Для серверної частини аплікації використаємо onion-архітектуру. Згідно з цим підходом в основі нашої аплікації знаходиться доменна модель, навколо якої надбудовуються додаткові рівні логіки. На одному рівні може знаходитись декілька незалежних шарів. При цьому важливо, що рівні мають доступ до всіх рівнів нижче. Зв’язок із нижчих шарів до вищих не дозволяється, *рисунок 2.9*

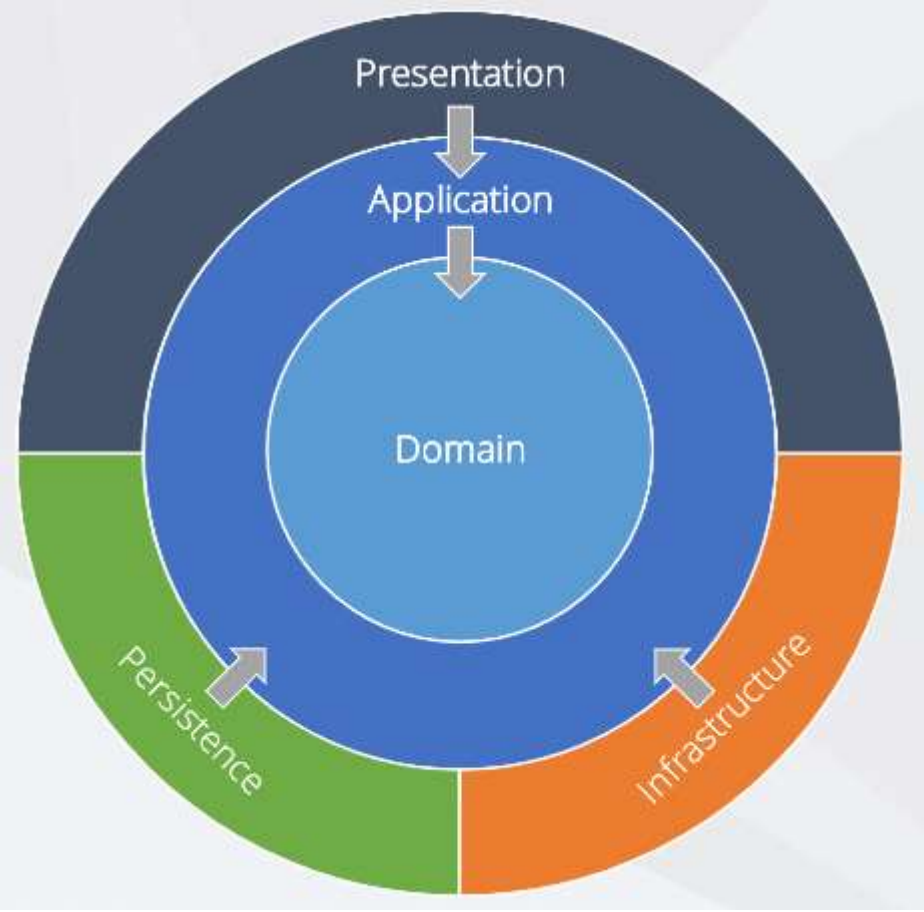

Рисунок 2.9 — Приклад взаємодії модулів при використанні onion-архітектури.

Інший підхід, який допоможе нам впоратись зі складністю предметної області це предметно-орієнтоване проєктування (англ. domain-driven design, DDD). Переваги DDD полягають в наступному:

а) концентрація основної уваги на предметній області;

б) створення програмних моделей, які відображують глибоке розуміння предметної області;

в) використання об’єктно-орієнтованого програмування для опису взаємодій між моделями предметної області;

Для того, щоб повноцінно розкрити можливості WebSocket протоколу реалізуємо подійно-орієнтовану модель. Згідно із заданою доменною областю клієнти повинні отримувати інформацію про будь-які зміни із вантажем у реальному режимі.

Підхід при якому ми зберігаємо не фінальний стан об’єкта, а зміни над ним називається Event sourcing. При звичайному підході ми зберігаємо лише фінальний стан об’єкта, а будь-які зміни над ним затирають його попередній стан. При реалізації Event sourcing підходу ми не маємо стану як такого, а лише набір подій. Кожна подія містить лише необхідні зміни, *рисунок 2.10:*

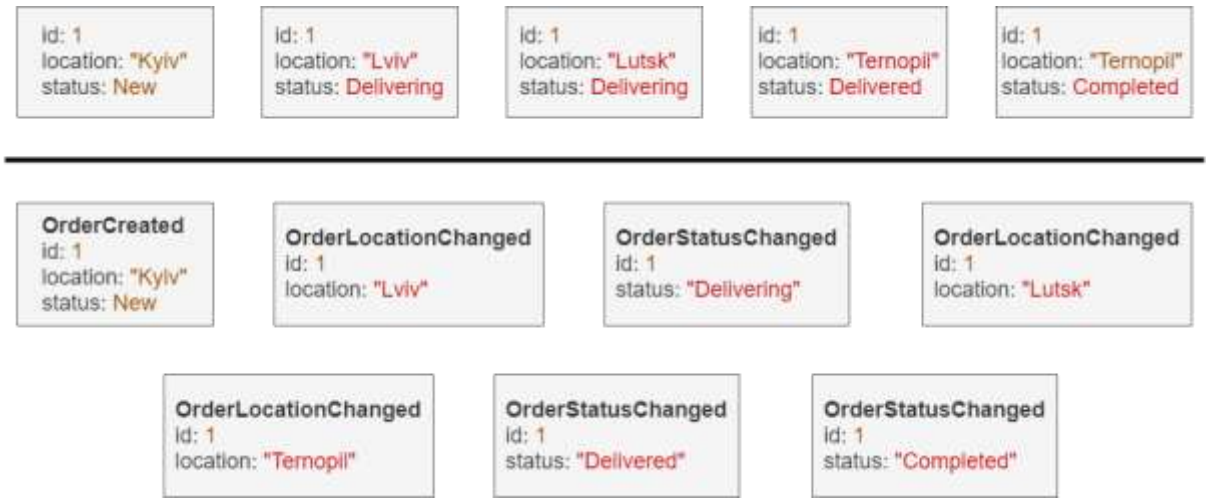

Рисунок 2.10 — Приклад звичної моделі збереження даних зверху, та подійно-орієнтована знизу відповідно.

Події не видаляються, а лише додаються, тобто, якщо потрібно скасувати якісь зміни, ми додаємо нову подію із даними, що компенсують зміни. Якщо потрібно отримати об’єкт з усіма його полями, необхідно застосувати усі події послідовно.

Також варто зазначити, що для підтримки масштабування та високого навантаження сервісу використовується CQRS підхід. CQRS (англ. command-query responsibility segregation) — це підхід, при якому дії із даними розділяються на команди та запити. Команди відповідальні за зміну, видалення чи редагування ресурсу, тобто будь-яку операцію, яка передбачає зміну стану об'єкту. В той час, як запити виконують лише читання даних. Таким чином ми зможемо розділити логіку аплікації та зменшити навантаження з інфраструктури. Використання даного підходу доцільно у випадках коли одна із систем (зазвичай читання) не справляється із навантаженням.

Також ми знаємо, що користувач повинний отримати повідомлення, яке містить номер накладної вантажу, якщо нове замовлення було створено. Повідомлення відправляються не так часто, та для того, щоб не ускладнювати основний сервіс винесемо цю логіку в окремий компонент. Даний компонент може працювати паралельно, а також при потребі незалежно масштабуватись.

2.4 Модель даних

Сформуємо базу даних. Основною сутністю буде замовлення (таблиця Order), яке містить вантажі (таблиця Cargo). Замовлення є основною сутністю, в той час, як вантаж є залежною, *рисунок 2.11*:

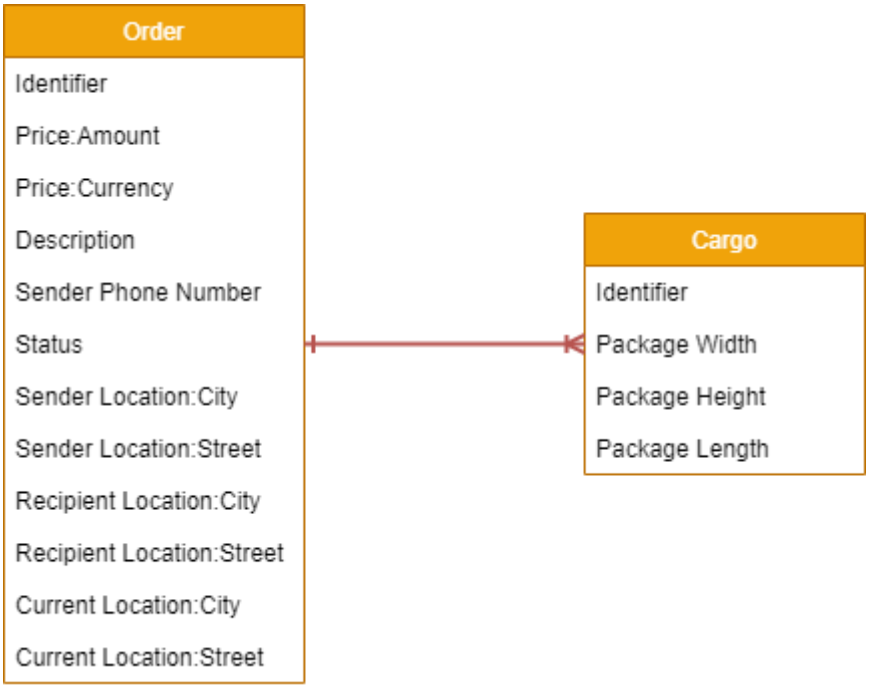

Рисунок 2.11 — Модель «сутність-зв'язок» предметної області.

Доменні події не мають структури та використовуються для того, щоб побудувати на їх основі стан наших сутностей, *рисунок 2.12:*

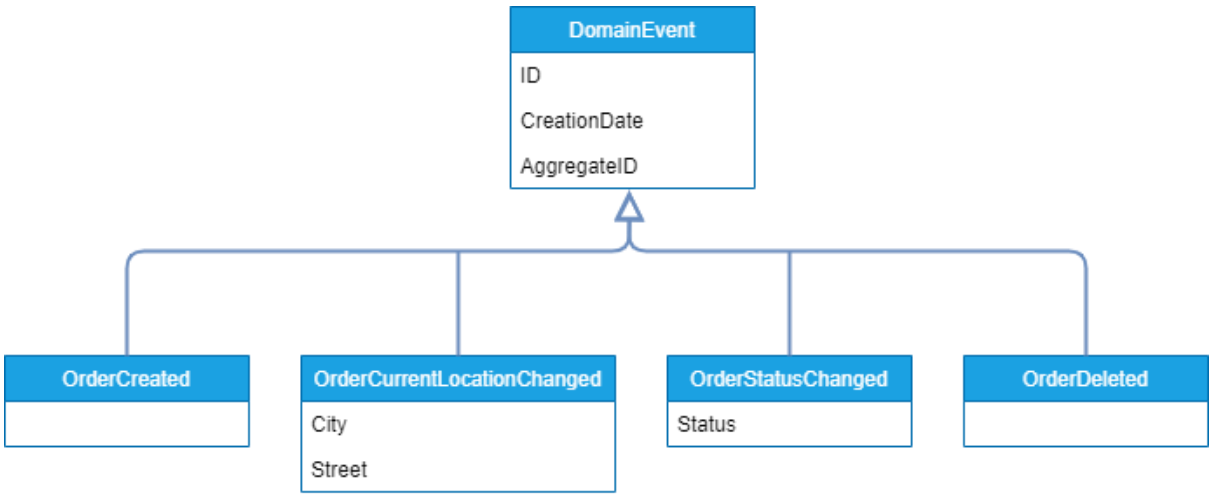

Рисунок 2.12 — Модель «сутність-зв'язок» подій предметної області.

Також нам потрібний спосіб для підтримки консистенції даних. Для цього використовуються так звані “інтеграційні” події, *рисунок 2.13*. Вони зберігаються перед надсиланням зі станом “в прогресі” та по успішному завершенню транзакції їх стан оновлюється на “відправлені”. Список таких подій корисний і тим, що дозволяє відстежити які сервіси були сповіщенні.

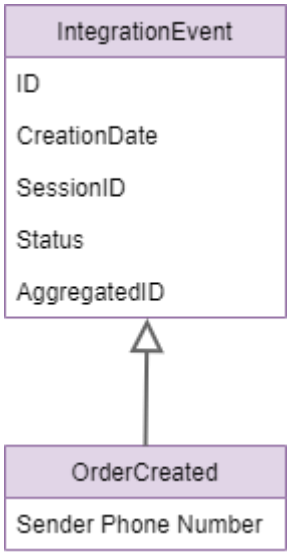

Рисунок 2.13 — Модель «сутність-зв'язок» зовнішніх подій.

Дана модель не є фінальною, але вона становить гарний початок.

2.5 Огляд

Отже, на цей момент архітектура додатків виглядає наступним чином:

а) Для побудови додатку клієнта використовується SPA підхід із використанням фреймворку Angular;

б) Для побудови основного сервісу аплікації використовується onion архітектура із використанням CQRS підходу;

в) Взаємодія між клієнтом та сервером відбувається за допомогою WebSocket протоколу у двосторонньому напрямку.

г) Як основне сховище використовується MongoDb. Це документно-орієнтована не реляційна база даних. Вона підходить, як для збереження доменних подій, що не мають схеми, так і для сутностей, тому що дозволяє містити основну та залежну сутність разом тим самим уникаючи перегляду багатьох таблиць.

д) Redis — сховище типу ключ-значення використовуються для підтримки консистентності даних між різними серверами. При горизонтальному масштабуванні основного сервісу, можуть виникати проблеми при роботі

з одними та тими самими даними. Використовуючи додаткове спільне сховище, яке містить ключі блокування дозволяє уникнути цих проблем

е) Також використовується спеціальний фоновий процес, який за допомогою асинхронної передачі подій чергою, надсилає Смс-повідомлення.

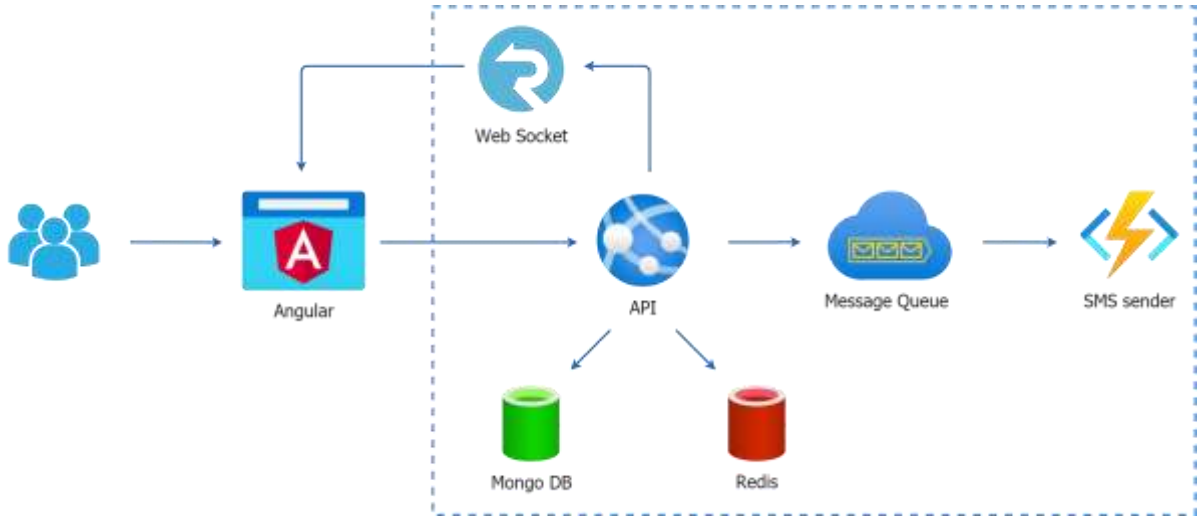

Рисунок 2.14 — Огляд архітектури системи.

Даної структури цілком достатньо аби розпочати розробку аплікації. Всі наступні компоненти будуть додані при потребі.

3 РОЗРОБКА ТА РОЗГОРТАННЯ

3.1 Побудова частини працівника

3.1.2 Створення замовлення

Як зазначалось раніше для працівника поштового відділення є можливість створити нове замовлення, та переглянути список усіх замовлень, *рисунок 3.1*:

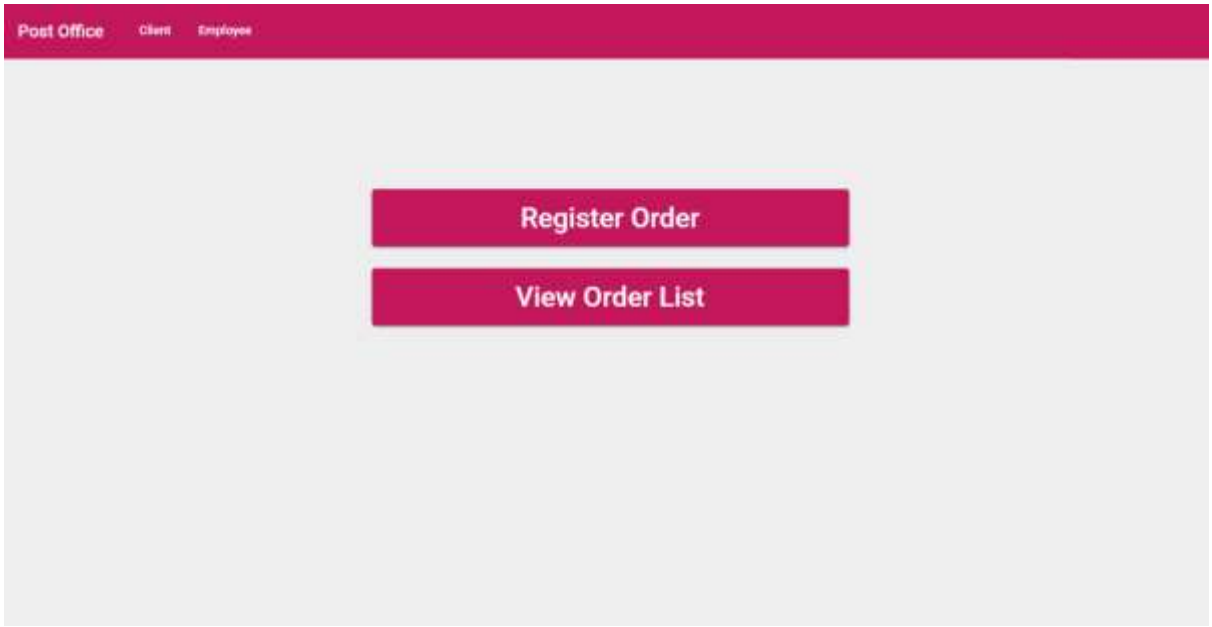

Рисунок 3.1 — Сторінка вибору дій для працівника.

Створення замовлення відбувається в декілька етапів. На першому етапі необхідно ввести інформацію про замовлення, адресу відправника та адресу отримувача, *рисунок 3.2:*

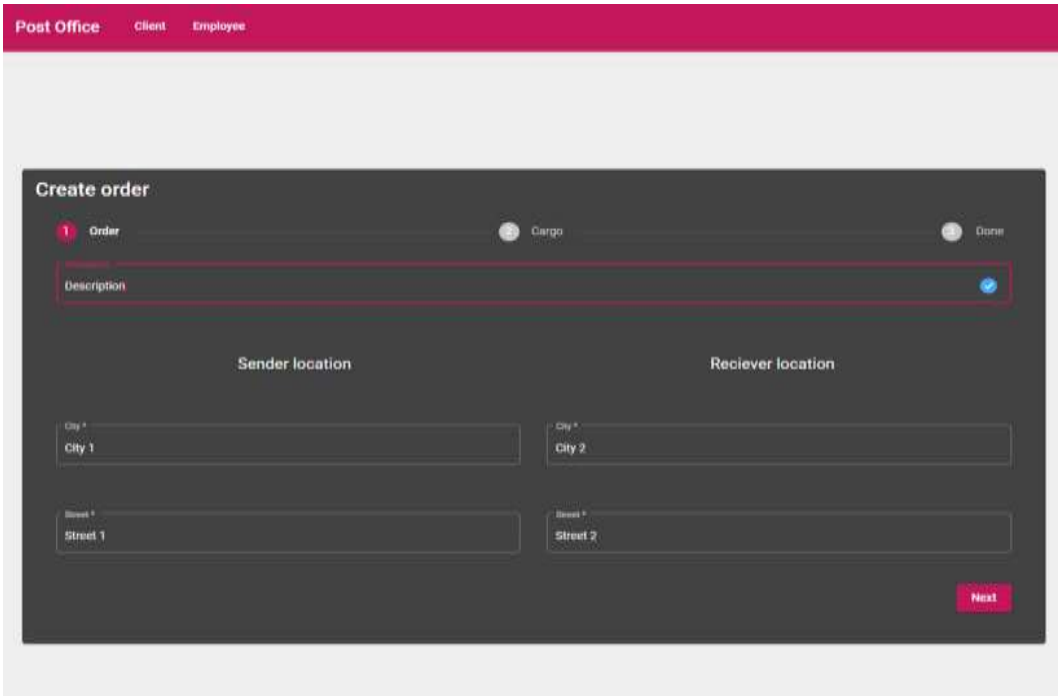

Рисунок 3.2 — Сторінка створення нового замовлення. Приклад додавання інформації про замовлення.

На наступному етапі можна додати новий вантаж, вказати інформацію його габарити, чи видалити випадково доданий, *рисунок 3.3:*

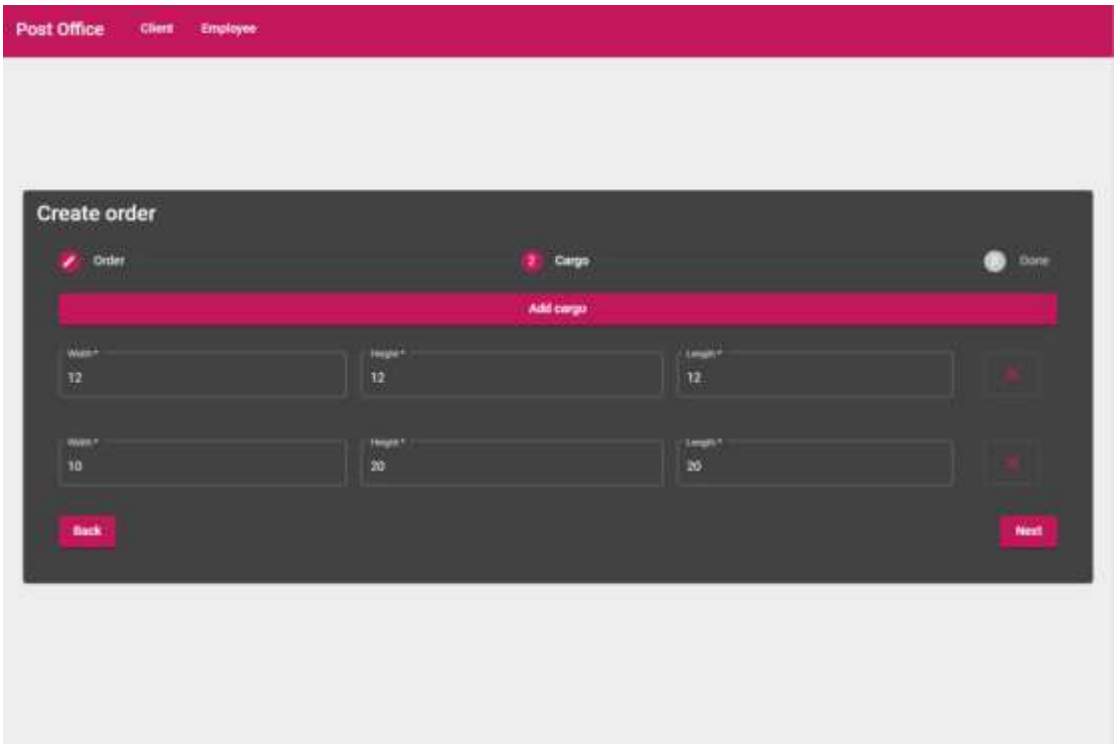

Рисунок 3.3 — Сторінка створення нового замовлення. Приклад додавання вантажів.

Останній етап вимагає підтвердження замовлення. Хоч і процес створення розбитий в декілька кроків, уся інформації зберігається на клієнтській частині до завершення усіх етапів.

На стороні сервера процес створення замовлення зачіпає декілька серверів. Розглянемо його детальніше за допомогою діаграми послідовностей, *рисунок 3.4:*

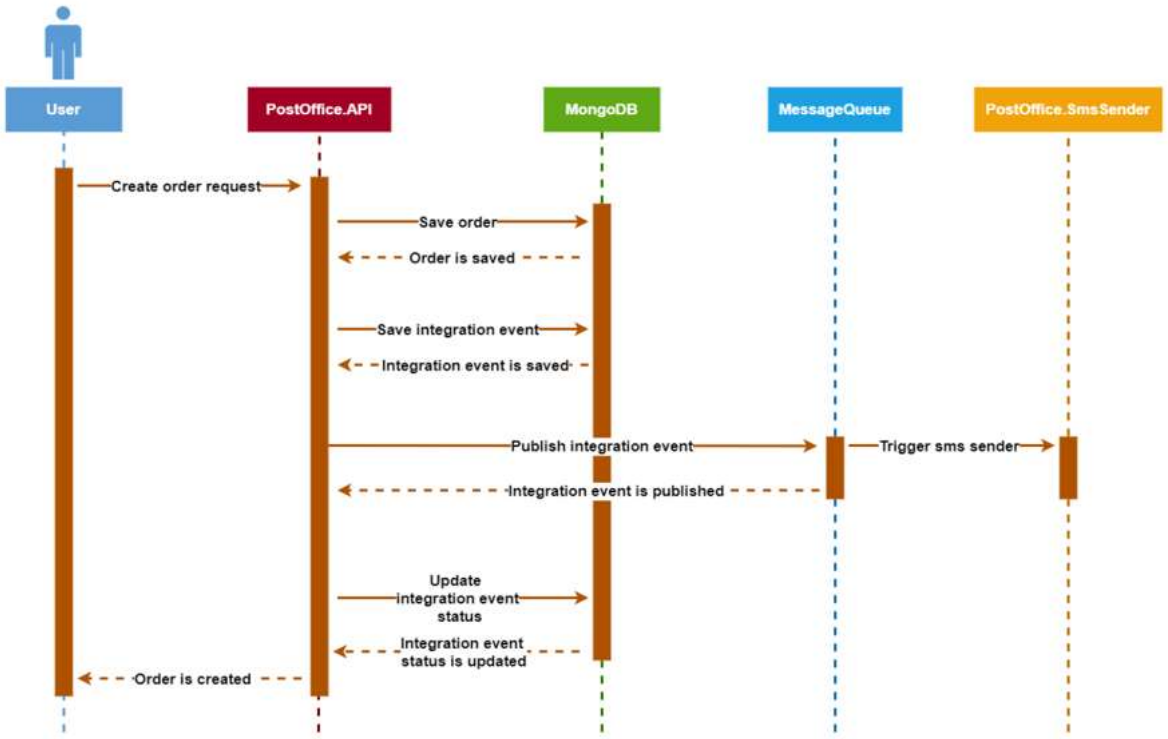

Рисунок 3.4 — Діаграма послідовностей взаємодії сервісів під час створення нового замовлення.

а) щойно користувач підтвердив замовлення, усі дані відправляються від клієнта на сервер;

б) сервер перевіряє валідність інформації та у випадку коректності даних створює запис у сховищі — MongoDb;

в) під час виконання логіки на попередньому етапі сутність “Замовлення” накопичувала набір подій. В межах виконання транзакції зі збереженням сутності, усі накопичені події, створені раніше, відправляються підписникам. Події можуть бути оброблений лише в межах того самого процесу. Для того, щоб і інші сервіси могли реагувати на зміни, події зберігаються в сховищі зі станом, який вказує, що їх необхідно відправити;

г) в сховищі відбувається пошук нових подій, та в разі їх наявності вони відправляються на спеціальний вузол (MessageQueue), який відповідальний за те, щоб проінформувати решта сервісів;

д) одна із таких подій містить інформацію про створення нового замовлення. Ця інформації необхідна для того, щоб відправити користувачу смс-повідомлення. Оскільки відправник повідомлень може бути навантажений даний процес відбувається в асинхронному режимі;

е) після успішної публікації події оновлюється її статус;

ж) як результат клієнт отримує інформацію про успішне створення замовлення.

З діаграми вище може здатись ніби процес збереження подій непотрібний і лише викликає додаткову складність в роботі алгоритму, та це не так. Усі кроки необхідні для підтримування узгодженості в системі.

Так, наприклад розглянемо можливі неполадки у системі те яким чином система на них реагує:

а) першим рівнем збоїв може виступати недоступність серверу. У такому випадку користувач отримає повідомлення про помилку;

б) аналогічна поведінка відбувається недоступності сховища;

в) якщо сховище стане недоступним після збереження сутності, але до збереження подій, тоді збереження сутності буде скасоване оскільки дані операції відбуваються в межах однієї транзакції. Це необхідно для того, щоб не втратити події;

г) у випадку коли не доступний вузол відправлення подій, тоді у них не змінюється статус, та події будуть опубліковані при можливості;

д) відправних подій працює в контексті відокремленому від сервера. Він відповідальний за успішну публікацію подій. Буде здійснено декілька спроб відправлення події сторонньому сервісу допоки він не відповість про успішне опрацювання події. Якщо кількість спроб буде вичерпано дане повідомлення буде збережене та адміністратори системи будуть сповіщені про похибку. Кількість спроб відправлення події обмежується для того, щоб не навантажувати сервіс і дати йому можливість відновити свою роботу;

е) останній етап відмови у системі може виникнути коли сховище перестає бути доступним після публікації подій. У такому разі хоч і події були надіслані їх статус не змінився, що означає їх повторне відправлення. Дана поведінка теж небажана оскільки може призвести до неправильної роботи сторонніх сервіси. Щоб цього уникнути вузол відправлення подій реалізований ідемпотентним шляхом. Ідемпотентність — це властивість при якій повторна дія над будь-яким об'єктом не змінює його результату;

Як бачимо, система є стійкою при появі неполадок на будь-якому етапі та здатна коректно працювати й відновити свій стан у разі потреби.

3.1.2 Видалення замовлення

У користувача також є можлива переглянути список усіх замовлень та при потребі редагувати або видалити довільний запис, *рисунок 3.5*:

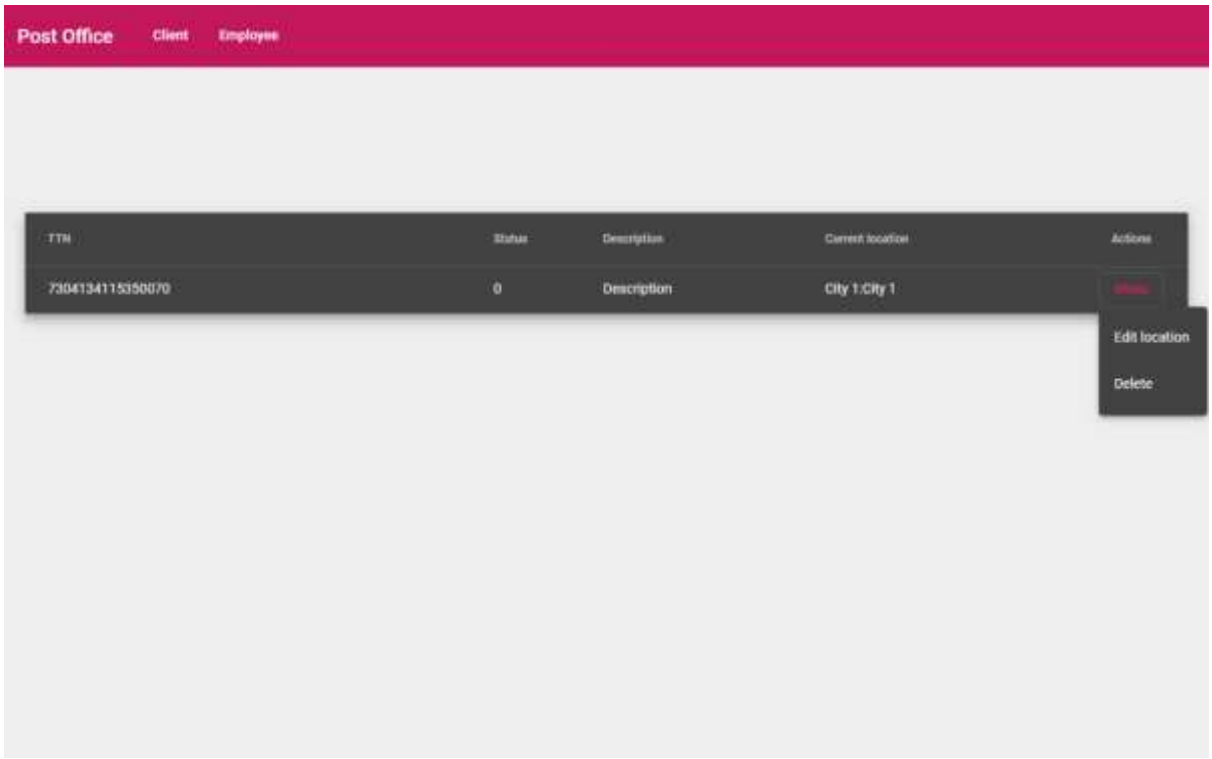

Рисунок 3.4 — Сторінка перегляду списку замовлень.

Видалення запису не зачіпає багато компонентів системи. Даний процес не вимагає високого рівня консистенції даних, а конкуренції з іншими користувачами системи не є критичною. Якщо два користувачі видаляють один і той самий запис

один із них отримає повідомлення, про успішне завершення операції, а інший про те що ресурс був видалений кимось іншим.

Що цікавого, це те що реалізація даної операції із використанням протоколу HTTP вимагає синхронної роботи через те, що сам протокол є синхронним. Своєю чергою WebSocket підтримує з'єднання активним, але сам сервер може виконувати роботу асинхронно. Таким чином, сервер відправляє запит в сховище і поки відбувається вибірка даних він може обробляти запити від інших користувачів. Щойно дані будуть знайдені, сервер зможе проштовхнути їх клієнту.

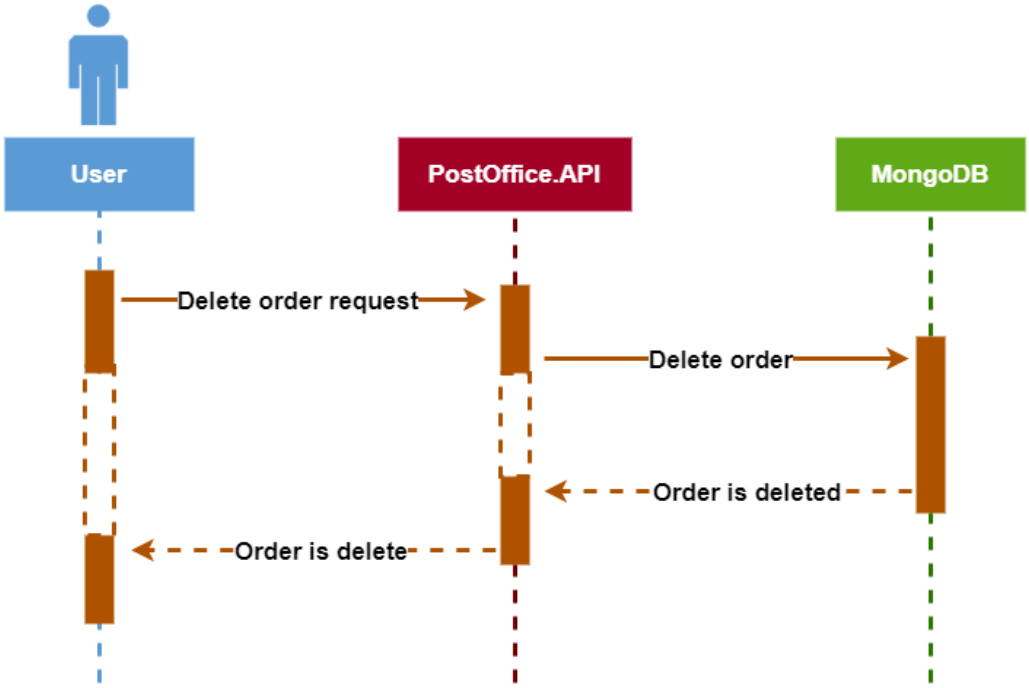

Рисунок 3.6 — Діаграма послідовностей взаємодії сервісів під час видалення замовлення.

3.1.3 Редагування замовлення

Процес редагування замовлення не такий простий як видалення, оскільки тут важливо зберігати дані в єдиній вірній формі при цьому варто пам'ятати, що не лише декілька користувачів можуть працювати з одним і тим самим записом, але й самих сервіс може бути декілька.

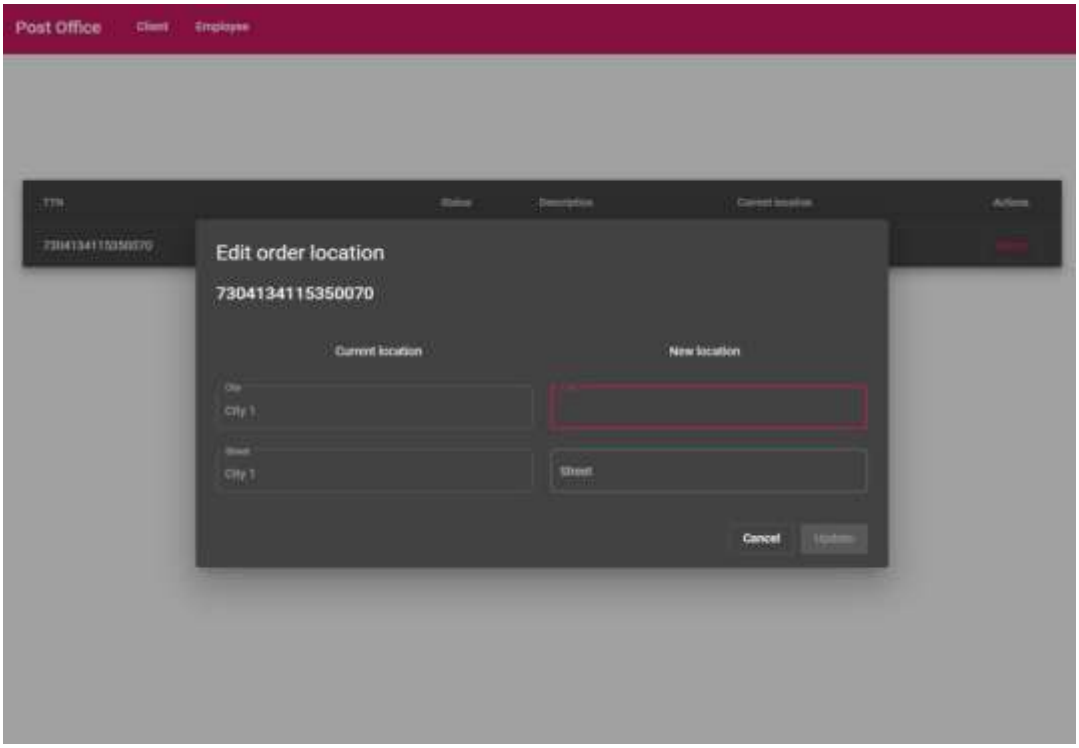

Рисунок 3.7 — Сторінка для редагування замовлення.

Розглянемо поетапно всі кроки завдяки діаграмі послідовностей, *рисунок 3.8:*

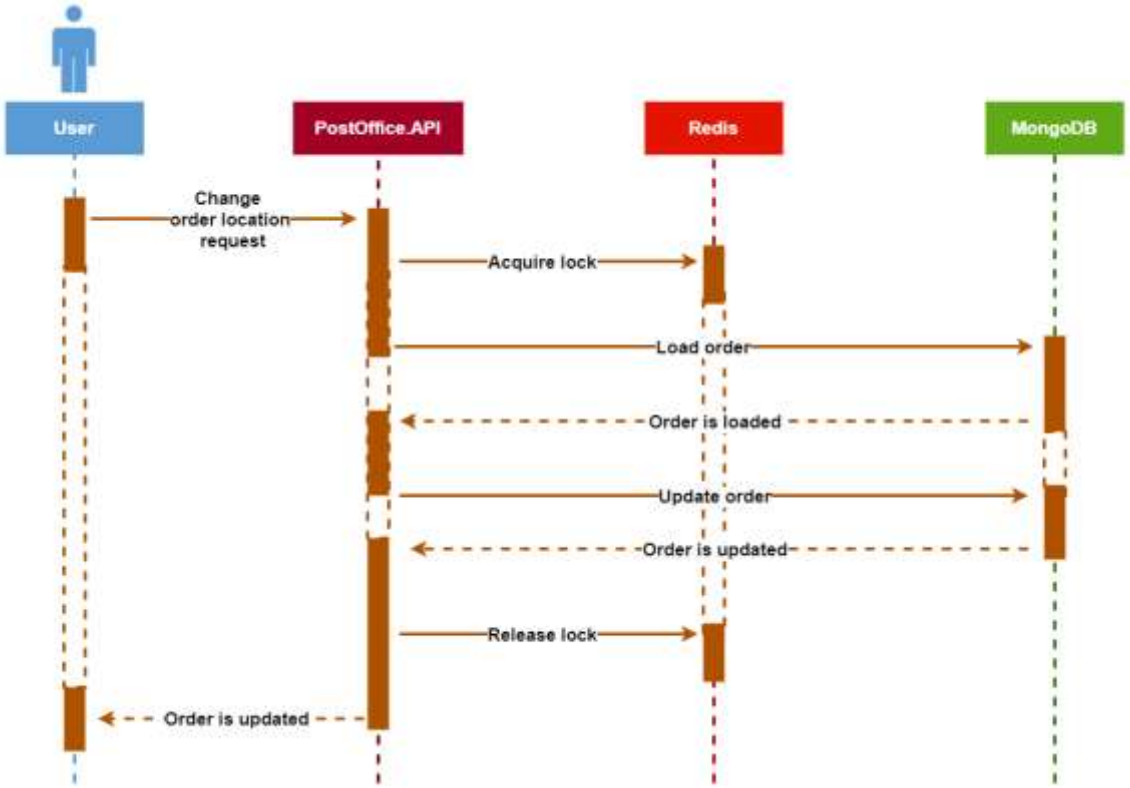

Рисунок 3.8 — Діаграма послідовностей взаємодії сервісів при редагуванні замовлення.

а) щойно користувач підтвердив зміну замовлення, усі дані відправляються від клієнта на сервер;
б) сервер блокує роботу із ресурсом створюючи запис у спеціалізованому сховищі — Redis;
в) сервер перевіряє валідність інформації та у випадку коректності даних витягує та редагує запис у сховищі — MongoDb;
г) після успішної зміни даних сервер видаляє запис із Redis сховища, та дозволяє іншим сервісам працювати із ним;
д) як результат усі клієнти отримують інформацію про оновлене замовлення;

Як і в раніше сервер працює в асинхронному режимі та може обробляти декілька запитів. Також важливо те що блокування ресурсу відбувається на локально, а з допомогою окремого сховища. Це необхідно, аби при горизонтальному масштабуванні декілька екземплярів серверу не могли конфліктувати один з одним.

3.2 Побудова частини користувача

3.2.1 Перегляд інформації про замовлення

Основним функціонал, який доступний користувачеві є перегляд інформації про замовлення, *рисунок 3.7*:

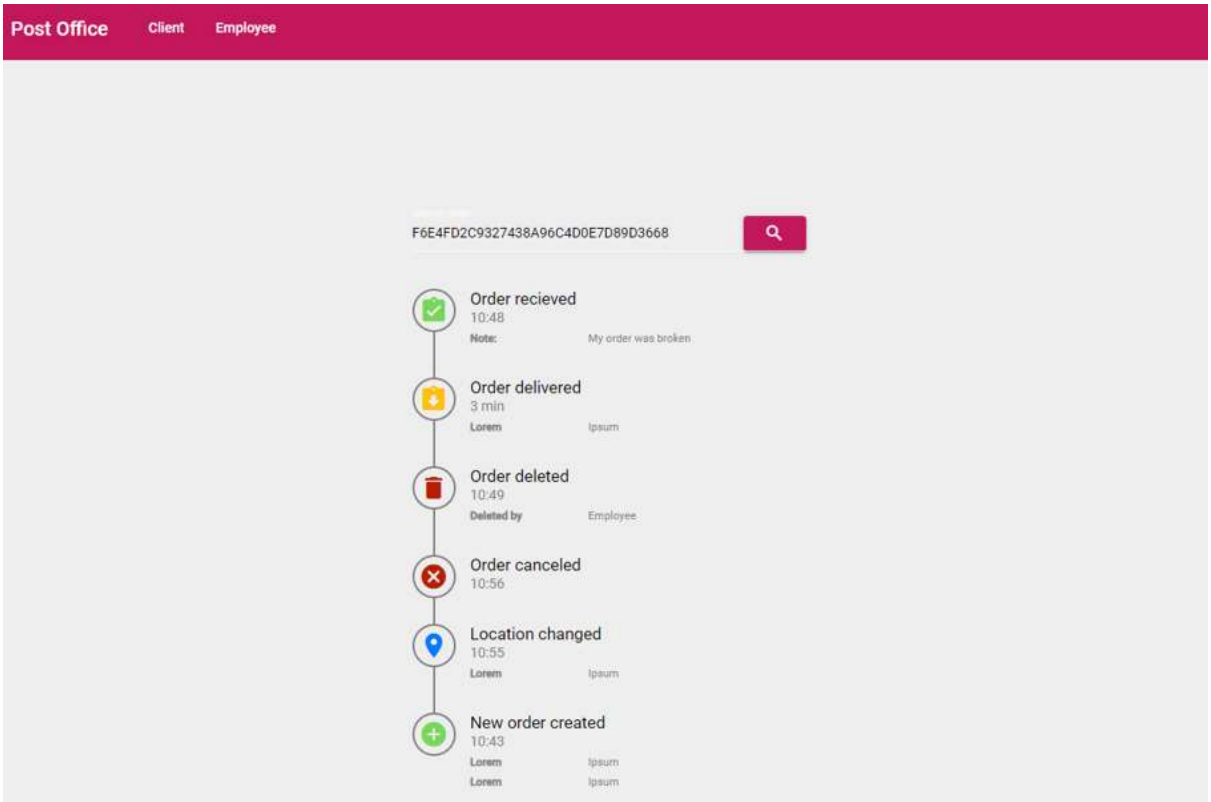

Рисунок 3.7 — Сторінка перегляду інформації про замовлення.

Важливо зазначити, що завдяки використанню протоколу WebSocket користувач отримує інформацію про будь-які зміни у реальному часі. Дана особливість значно покращує досвід користувача, оскільки дозволяє не перезавантажувати постійно сторінку, тим самим уникнувши безлічі зайвих операцій.

3.3 Розгортання аплікації

Для того, щоб до даної системи отримали доступ інші користувачі та аби мати змогу провести тестування аплікації в реальних умовах її необхідно розгорнути на сервері. Одним із таких середовищ є Azure — хмарна платформа, що призначене для розгортання онлайн додатків.

Платформа Azure дозволяє розділити контейнери для функціонування аплікації у вигляді так званої інфраструктури, та наповнити їх вміст при можливості. Це корисно в тому випадку, коли необхідно спроєктувати компоненти та взаємодію між ними без наявності самого програмного продукту.

Спробуємо проаналізувати які компоненти входять в систему:

а) центральним компонентом системи є API сервер взаємодія із яким відбувається по WebSocket протоколу;

б) основне сховище даних у вигляді MongoDb;

в) Redis для підтримання консистентності даних;

г) функції Azure – безсерверні застосунки, для виконання асинхронних операцій. За їх підтримку відповідальне хмарне середовище, яке динамічно розгортатиме їх при потребі.

д) аплікація-клієнт, яка містить вебінтерфейс взаємодії із додатком. Він може бути розгорнутий в тому самому вузлі що й API-сервер та виконуватись в окремому процесі. Такий підхід дозволить нам зекономити на ресурсах, щоправда, взаємодія між клієнтом та сервером буде швидшою, що може вплинути на апробаційний етап. Тому розгорнемо його як окремий компонент, що краще симулює реальні умови.

Отже, отримаємо наступну діаграму розгортання, *рисунок 3.8*:

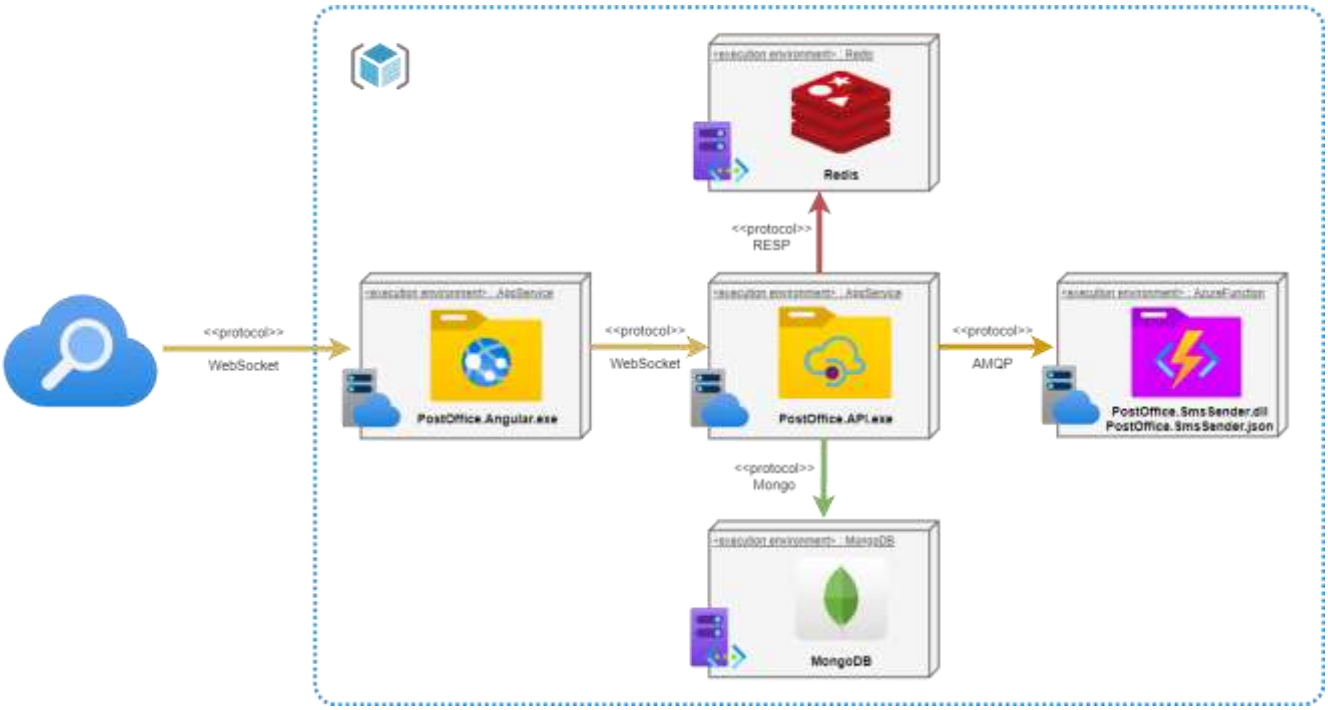

Рисунок 3.8 — Діаграма розгортання аплікації.

Як зазначалось вище, Azure надає лише місце для розміщення аплікація. Для того, щоб спростити процес створення необхідної інфраструктури скористаємось програмним забезпеченням під назвою Terraform. Цей продукт дозволяє описати необхідні компоненти за допомогою декларативної мови програмування. Даний підхід також часто називають Інфраструктура як код.

Обране рішення має декілька переваг над звичайним підходом:

а) налаштовування необхідного обладнання власноруч чи з допомогою інтерактивних інструментів часто приводить до помилок;

б) команди опису інфраструктури можуть перебувати в системі контролю версій;

в) простіше вносити зміни, до наявних компонентів;

г) інфраструктура як код може бути частиною неперервної інтеграції.

Окрім того, щоб описати необхідну інфраструктуру, необхідно також налаштувати процес неперервного розгортання. Тобто описати правила згідно з якими програмний продукт буде розгортатись в хмарному середовищі.

Платформа Azure надає інструмент для автоматизації неперервного розгортання — Pipelines. Для їх налаштування, необхідно описати набір правил мовою розмітки YAML. Таким чином при кожній зміні в системі контролю версій (https://github.com/iamprovidence/PostOffice) Azure буде аналізувати ці правила, та оновлювати розгорнений продукт.

4 АПРОБАЦІЯ

Одним із важливих етапів розробки є апробація — перевірка на практиці, в реальних умовах теоретично побудованих методів.

Протестуємо аплікацію у наступних сферах:

а) продуктивність;

б) ресурсозатратність;

в) стійкість до навантажень;

г) стійкість до загроз безпеки.

4.1 Перевірка продуктивності аплікації

Як зазначалось раніше WebSocket протокол вимагає меншу кількість TCP з'єднань ніж HTTP, а отже, запити, що його використовують повинні виконуватись швидше.

Спробуємо заміряти продуктивність за допомогою бібліотеки BenchmarkDotNet. Задана бібліотека зробить 25 запитів для кожного із протоколів та визначить середнє значення тривалості обробки запиту. Варто зазначити, що виміри відбуваються при розгортанні аплікації в режимі випуску, а отже, код оптимізований компілятором та не містить надлишкових інструкцій, що притаманні режиму розробки. Розглянемо отриманий результат вимірів, *рисунок 4.1*:

Method	Mean	Error	StdDev	Median	Rank	Gen 0	Allocated
SignalRTest	360.4 us	54.58 us	160.9 us	240.3 us	1	0.9766	3 KB
HttpTest	501.9 us	64.73 us	190.9 us	586.7 us	2	0.9766	3 KB

Рисунок 4.1 — Результати вимірів продуктивності.

Серед наведених даних можна побачити заміри в наступних категоріях: середня тривалість виконання, похибка, середнє квадратичне відхилення, медіана, порядок виконання та інформацію про затратність пам'яті, відсоток об'єктів, що були очищені під час обробки збирачем сміття першого покоління та кількість виділеної пам'яті відповідно.

Хоч дана бібліотека дозволяє детально відстежити ресурсозатратність, та, як бачимо із вищенаведених даних використання кількості сокетів різними протоколами настільки незначні, що не піддаються вимірам.

Окрім цього знехтуємо результатами похибки через те, що вони залежать від багатьох непередбачуваних чинників, а також збігаються в обох методах.

Цікавим показником є середня тривалість виконання запитів. Як бачимо, теоретичні дані підтвердились на практиці та WebSocket протокол працює майже в півтора раза швидше за HTTP.

Може здатись, що різниця незначна, адже вона вираховується в мікросекундах, та навантаження сучасних серверів в середньому становить 15 мільйонів запитів в місяць. Наразі розробники борються за кожну долю секунди та намагаються оптимізувати незначні дрібниці. Бувають випадки коли через обмеженість середовища розробки доводиться змінювати мови програмування з однієї на іншу. Тож використання двостороннього протоколу із більшою швидкістю виконання виглядає лише подарунком.

4.2 Перевірка ресурсозатратності аплікації

Наступним етапом апробації є перевірка ресурсозатратності заданої архітектури та порівняння її зі стандартним підходом використовуючи Http.

Як зазначалось раніше бібліотека BenchmarkDotNet не здатна зафіксувати незначні зміни в пам’яті, тому скористаємось механізмом для перевірки навантаження вбудованим у середовище розробки VisualStudio.

Водночас при аналізі варто враховувати деякі особливості системи, що тестуємо:

а) платформа dotNET реалізує механізм автоматичного очищення пам’яті. Його робота є недетермінована, та залежить від багатьох факторів: навантаження системи, час останнього запуску тощо. Робота цього механізму може вплинути на заміри;

б) обидва протоколи HTTP та WebSocket використовують незначну, за сучасними мірами, кількість ресурсів, що ускладнює виміри та як результати непомітні, а похибка доволі висока;

в) інструменти для вимірювання ресурсозатратності, окрім необхідної інформації за кількістю зайнятої пам’яті вебсокетами вимірюють також затратність ресурсів витрачених на підтримку роботи аплікації;

Розглянемо кожну із проблем детальніше, та спробуємо знайти рішення її розв’язку.

Першою проблемою є механізм очищення пам’яті, що реалізований платформою, на якій ведеться розробка. Як зазначалось раніше, він є недетермінованим, з чого випливає, що неможливо передбачити в який момент часу відбудеться процес очищення. Здається, що можна спровокувати запуск його роботи з визначеним наперед інтервалом, щоб уникнути автоматичної роботи алгоритму, але це спричиняє лише більше проблем, бо тепер наша аплікація витрачає ресурси також на те, щоб контролювати очищення. При цьому HTTP та WebSocket можуть використовувати різну кількість пам’яті, а очищення зайвих об’єктів, лише зіпсує результати. Єдиним правильним рішенням буде відключити автоматичний механізм очищення.

Наступна проблема полягає в тому, що обидва протоколи побудовані поверх TCP з’єднання із додатковою надбудовою зі своєї сторони. Такий тип взаємодії є стандартом протягом багатьох років. Якщо ще за часів появи, цієї технології, вона і могла створити навантаження на комп’ютери, то наразі, сучасні обчислювальні машини здатні підтримувати до сотні активних з’єднань без будь-яких помітних проблем. Таким чином, навіть, бібліотека BenchmarkDotNet, що призначена для вимірів низькорівневих операцій, не здатна побачити будь-якого навантаження. Щоб розв'язувати цю проблему будемо перевіряти на ресурсозатратність не одну операцію, а декілька тисяч. Отже, внаслідок великого навантаження ми зможемо перевірити ресурсозатратність протоколів.

Остання проблема полягає в тому, що заміри відбуваються для всієї аплікації, а не лише для мережевої взаємодії. А отже, порівнюючи результати, вони не будуть

точними, а лише показуватимуть різницю між протоколами. Для того, щоб виміряти ресурсозатратність кожного із протоколів необхідно, зробити додатковий замір, виконання самої аплікації, без передачі даних мережею. А тоді отриманий результат відняти від результатів отриманих при вимірі кожного із протоколів.

Результати отриманих вимірів можна побачити на *рисунку 4.2:*

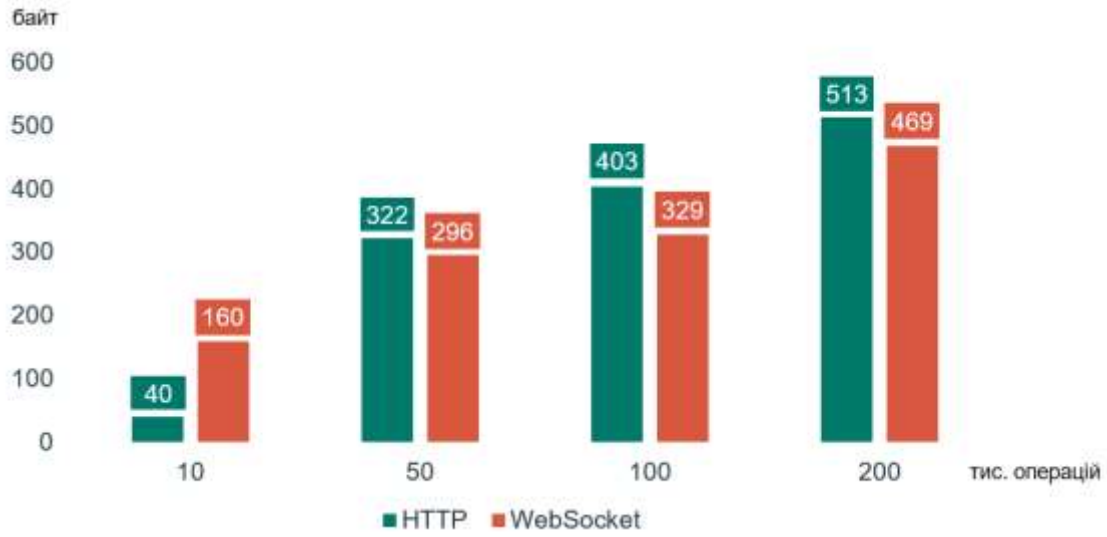

Рисунок 4.2 — Результати вимірів ресурсозатратності.

Як бачимо, при незначній кількості операцій WebSocket програє HTTP майже в чотири рази. Та при збільшенні кількості операцій, їх ресурсозатратність вирівнюється, а згодом переваги переходить від одного до іншого. Може здатись ніби й цього разу, WebSocket є фаворитом, та насправді останні два виміри були зроблені лише для теоретичної перевірки. На практиці ж доволі рідко доведеться зіткнутись із подібними навантаженням.

Отже, ресурсозатратність WebSocket протоколу є вищою, або ж рівною ресурсозатратності HTTP протоколу.

4.3 Перевірка аплікації на стійкість до навантажень

Перевірка на стійкість до навантажень — це вид тестування, при якому відбувається збір показників, та визначення продуктивності аплікації та часу відгуку програмного забезпечення з метою встановлення вимог.

Для того, щоб провести подібного роду тести необхідно симулювати поведінку аплікації в реальних умовах. Для цього скористаємось бібліотекою K6. Даний інструмент дозволяє описати кількість віртуальних користувачів, які будуть надсилати запити на сервер та період протягом якого їх кількість буде сталою чи змінюватись, *рисунок 4.3*.

Рисунок 4.3 — Приклад симуляції запитів віртуальних користувачів здійснених бібліотекою K6.

Такий тип тестування дозволить нам визначити, яка максимальна потужність нашої системи з точки зору користувачів та пропускної здатності.

Одним із нефункціональних вимог до системи, було визначено, що аплікація повинна підтримувати 1000 одночасних користувачів, що і будемо вважати верхньою межею вимірювання.

Та для порівняння перевіримо час відгуку системи при 10, 100, 500 та 1000 запитів. Результати наведено на *рисунку 4.4*:

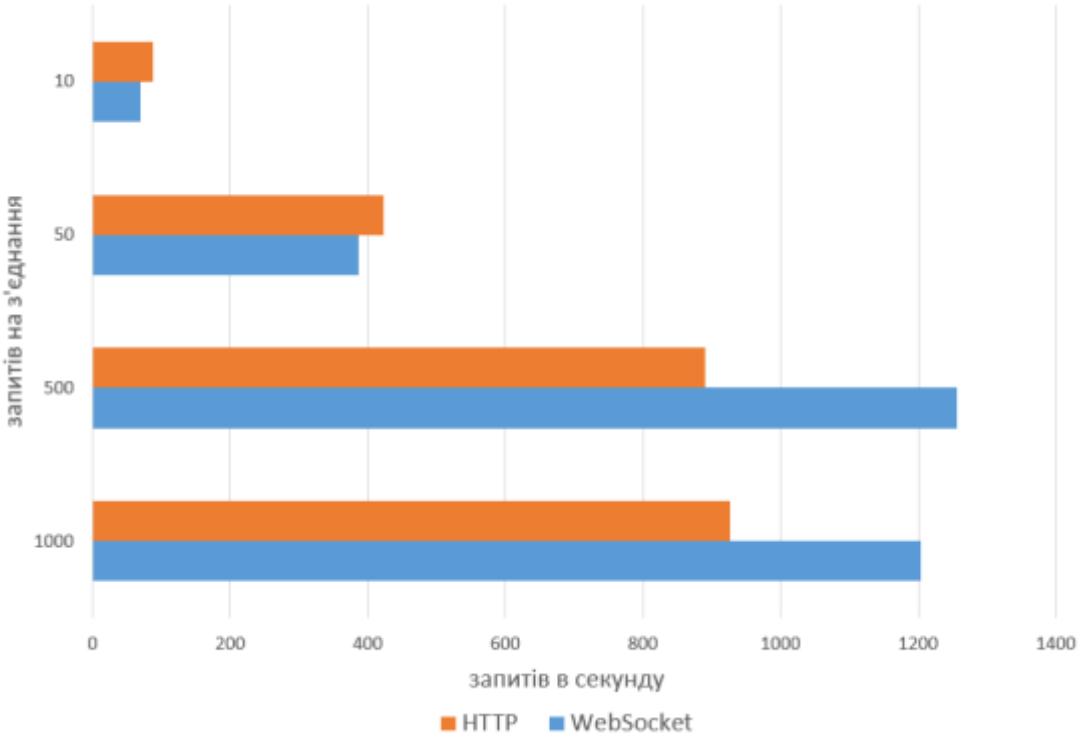

Рисунок 4.4 — Результати вимірів стійкості до навантаження.

Як можна побачити, при десяти активних користувачах WebSocket протокол програє по швидкості HTTP. Цю саму різницю можна помітити й при 100 активних з’єднаннях. Що цікаво, це те що при великому навантажені WebSocket здатний краще підтримувати стабільність роботи.

Як ми бачили раніше, даний протокол працює швидше і використовує менше пам’яті при великих навантаженнях, що і приводить до цього результату. Окрім цього HTTP протокол вимагає постійного надсилання заголовків, чого не робить WebSocket.

Хоч і перевірка відбувалась із віртуальними користувачами, в реальних умовах при наявності веббраузера дана статистика може значно погіршитись, оскільки більшість популярних браузерів мають обмеження на кількість паралельних запитів здійснених із використанням протоколу HTTP в той час, як немає жодного обмеження на кількість переданих повідомлень по WebSocket протоколу.

Важливо зазначити, що проблема навантаження на сервер вирішується за допомогою масштабування системи. Існує декілька підходів для масштабування: вертикальний та горизонтальний.

При вертикальному масштабуванні, ми збільшуємо кількість доступних ресурсів для аплікації шляхом покращення апаратного забезпечення. Такий підхід простий, оскільки не вимагає змін в коді програми. Та він швидко вичерпує себе, оскільки сильно залежить від розвитку технологій та максимально доступних обчислювальних можливостей сервера.

Альтернативою вертикального масштабування є горизонтальне, при якому збільшується кількість серверів. Даний підхід складніший в реалізації, через потребу в синхронізації даних між різними серверами, та при цьому він дозволяє здійснювати паралельні обчислення.

Проблема в синхронізації серверів вирішується за допомогою вузла синхронізації, додаткового сервера, який буде містити спільні дані. Недолік такого підходу полягає, в тому, що даний об’єкт повинний бути єдиний в системі та часто він стає вузьким місцем, який можна масштабувати лише вертикально.

Як можна побачити, WebSocket протокол куди стійкіший до навантажень. Та розглянемо проблему, масштабування, що виникає з WebSocket протоколом, та не з HTTP.

Нехай аплікація розгортається на декількох серверах, до яких під'єднані різні клієнти незалежно один від одного. При змінах даних на одному сервері варто сповістити всіх клієнтів, *рисунок 4.5*.

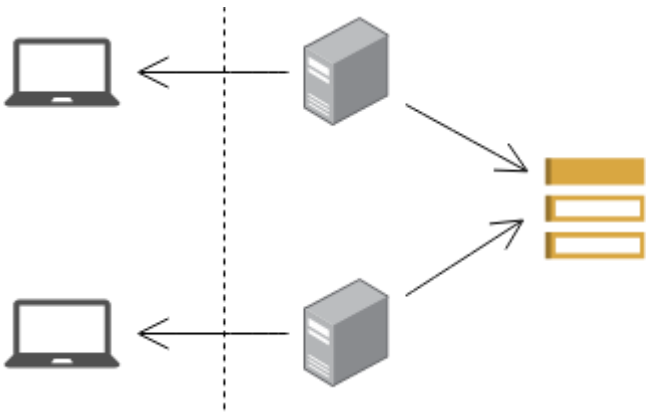

Рисунок 4.5 — Проблема горизонтального масштабування WebSocket протоколу.

З використанням HTTP протоколу, ми можемо звертатись по дані до довільного сервера, але із WebSocket нам потрібний додатковий журнал змін, що слугуватиме вузлом синхронізації. Такі інтеграційні події опублікуються зберігаються в спільну базу, а сервери реагують на зміни та сповіщати своїх клієнтів.

Отже, хоч і WebSocket має кращу пропускну здатність, більшу швидкість та вищий поріг навантаження, варто розуміти, що стійкість системи, до перенесення стресових навантажень залежить від багатьох факторів, таких як можливість масштабування системи та обраних компонентів і взаємодії між ними.

4.4 Перевірка аплікації на стійкість до загроз безпеки

Для того, щоб перевірити аплікацію на стійкість до загроз безпеки скористаємось продуктом Threat Modeling Tool. Він дозволяє описати компоненти системи та взаємодію між ними, та на основі цих даних визначає загрози та шляхи їх вирішення. Варто зазначити, що порівнюючи використовується захищена версія обох протоколів із використанням SSL.

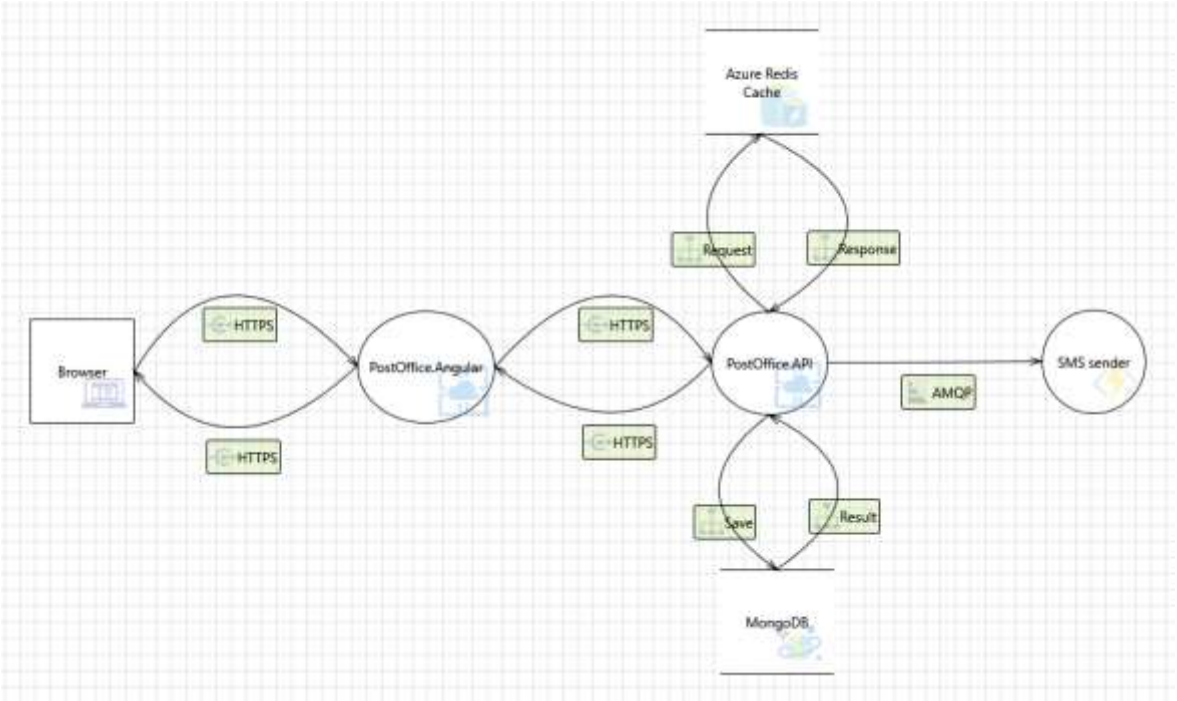

Рисунок 4.6 — Схема моделі загроз.

Обидва протоколи побудовані на основі TCP з’єднання, тож ряд загроз спільних для них обох можна відкинути.

Щобільше, браузер не ініціалізує WebSocket протокол при першому з’єднані, а відправляє HTTP запит із заголовком на запит про можливість оновлення. З цього випливає, що всі проблеми притаманні HTTP протоколу присутні й у WebSocket.

Однією із найпоширеніших загроз є атака на відмову в обслуговуванні, або ж DoS атака. Це один із найпоширеніших методів нападу на сервер за допомогою великої кількості зовнішніх запитів, з метою вивести його з ладу. Хоч WebSocket протокол має вищий поріг стійкості в умовах великих навантажень, та проблема DoS атаки поширеніша аніж в HTTP.

Основною причиною цьому є підтримка постійного з’єднання. Якщо з’єднання розірване, для його відновлення необхідно реалізувати алгоритм, що буде перевіряти чи сервер доступний із певною періодичністю.

Наївна реалізація цього алгоритму намагається встановити з’єднання із певним інтервалом, що лише більше навантажує сервер DoS атаками та не дозволяє йому відновити своєї роботи.

Куди кращим рішенням буде додати реалізацію випадкового відхилення періодичності з якою відновлюється з’єднання. Таким чином клієнти будуть намагатись звертатись до сервера у різні проміжки часу, та загальне навантаження буде нижчим.

Також можна скористатись рішенням, що підходить і для HTTP — обмеження кількості запитів та їх швидкості. Для того, щоб застосувати обмеження швидкості варто врахувати наступні вимоги:

а) причину несправності клієнта;

б) кількість трафіку, що отримує та надсилає клієнт;

в) навантаження сервера;

Також варто зазначити, що більшість протоколів авторизації/автентифікації такі як OpenId, WS-Federation, OAuth використовують у своїй роботі HTTP, оскільки він більш захищений та є стандартом, що підтримується усіма браузерами.

ВИСНОВОК

Отже, у рамках цієї роботи було реалізовано весь процес створення веб додатку із використанням вебсокет протоколу. Аплікація являє собою вебпрограму з об'єктноорієнтованим доступом до бази даних. Для написання системи мені знадобилися різноманітні інструменти на зразок Active Server Page, Entity Framework, MongoDb, Terraform тощо.

Також під час розробки, я переконався, що розробка вебсервера із використанням вебсокет протоколу не підходить для всіх типів додатків. Задана архітектура покликана зробити процес взаємодії із вебсторінками наближеним до використання нативних додатків. Вона розв'язує проблеми інтерактивності та підходить для розробки великих проєктів у яких і так присутній WebSocket протокол, але зовсім не виправдовує себе для незначних аплікацій.

Основними переваги використання даного підходу є:

а) інтерактивна взаємодія. Досвід взаємодії з аплікацією стає приємнішим та природнішим;
б) синхронізація даних між клієнтами. Усі користувачі даної системи мають завжди актуальні дані;
в) уніфікований код. Не потрібно використовувати різні підходи для роботи із HTTP та WebSocket протоколом тому, що використовується лише один протокол.

Серед недоліків можна зазначити наступне:

а) складність розробки. Якщо вам потрібно швидке рішення (невеликий додаток, стислі терміни), то даний підхід лише ускладнить розробку. Оскільки він не є розповсюдженим, важко знайти бібліотеки та готові рішеннями, якими можна скористатись. Більшість інфраструктурного коду доводиться писати власноруч;
б) складність підтримки. Наразі доступна незначна кількість інструментів для налагодження коду, а серед доступних може не виявитись необхідного;

в) узгодженість даних. Вебсокет протокол дозволяє сповіщати клієнта про зміни даних на сервері. При цьому, якщо важлива цілісність даних варто обробляти випадки, коли клієнт пропускає події, або не здатний їх коректно обробити;

Результати апробації показали, що WebSocket протокол має кращі показники щодо продуктивності та гірші в категорії ресурсозатратності. Він стійкіший до навантажень, але гірше масштабується. Також при його використанні необхідно затрачати додаткових зусиль для уникнення DoS атак при розривах з'єднання і спробі їх відновлення.

Для виконання даної задачі можна було скористатись і іншими технологіями. Одним із покращенням в процесу неперервної інтеграції могло стати використання Pulumi замість Terrafor. Це програмне забезпечення, що дозволяє описувати інфраструктуру як код за допомогою улюбленої мови, наприклад C#, що могло пришвидшити етап розгортання проєкту.

Також варто зазначити, що протокол HTTP також дозволяє надсилати події з сервера клієнтові по відкритому TCP з'єднанню за допомогою технології Server Send Event. Недоліком цього підходу, є те що з'єднання є односпрямованим і вимагає багато ресурсів на його підтримку.

Мета роботи була досягнута – дослідження переваг та недоліків створення додатків із використанням вебсокет протоколу. Процес створення системи складався з багатьох етапів, таких, як аналіз різноманітних наявних інструментів та технологій, продумування функціоналу, структури, розробки архітектури, інтуїтивно зрозумілого зовнішнього вигляду вебсайт та практичної реалізації. Окрім цього аплікація була розгорнута на хмарному сховищі Azure із застосуванням практик DevOps.

Наразі WebSocket протокол є хорошим рішенням, для того, щоб клієнт міг отримати інформацію від сервера, але він погано підходить для повноцінної розробки оскільки існує мала кількість інструментів для розробки та налагодження аплікації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Історія розвитку вебсервісів. — Режим доступу: http://web20mikitchak.blogspot.com/p/blog-page_26.html
2. Вебтехнології. Їх різновиди та функції. — Режим доступу: http://sites.znu.edu.ua/webprog/lect/1170.ukr.html
3. Використання Web-технологій в освіті та науці. — Режим доступу: https://sites.google.com/site/navcalnapraktikakitvoin/lekciie/lekcia-veb-tehnologiie
4. Andrew Troelsen. Pro C# 7: With .NET and .NET Core. — Minnesota, USA, 2017. – 1372 c.
5. Difference between HTTP and WebSocket (HTTP 2.0) — Available from: https://developerinsider.co/difference-between-http-and-http-2-0-websocket/
6. Connection id when calling SignalR Core hub method from controller. — Available from: https://stackoverflow.com/questions/50367586/connection-idwhen-calling-signalr-core-hub-method-from-controller
7. How to get SignalR user connection id out side the hub class. — Available from: https://stackoverflow.com/questions/19447974/how-to-get-signalr-userconnection-id-out-side-the-hub-class
8. How to build a real time notification system using SignalR Core. — Available from: https://medium.com/@NanoBrasca/how-to-build-a-real-time-notificationsystem-using-signalr-core-1fd4160454fa
9. Real-Time is feature your next web app needs. — Available from: https://alexyakunin.medium.com/features-of-the-future-web-apps-part-1-e32cf4e4e4f4
10. Onion architecture. — Available from: https://blog.ploeh.dk/2013/12/03/layersonions-ports-adapters-its-all-the-same/
11. SignalR documentation. — Available from: https://docs.microsoft.com/ruru/aspnet/core/signalr/hubs?view=aspnetcore-3.1

12. Redis. Replication vs Sharding. Sentinel vs Cluster. — Available from: https://rtfm.co.ua/redis-replikaciya-chast-1-obzor-replication-vs-sharding-sentinel-vs-cluster-topologiya-redis/
13. WebSocket. — Available from: https://uk.wikipedia.org/wiki/WebSocket
14. Microsoft .NET: Architecting Applications for the Enterprise, Second Edition Dino Esposito Andrea Saltarello. USA, 2015. — 418 с
15. Migrating your REST APIs to HTTP/2: Why and How? — Available from: https://blog.usejournal.com/migrating-your-rest-apis-to-http-2-why-and-how-8caee7d798fb
16. Lock (computer science) — Available from: https://en.wikipedia.org/wiki/Lock_(computer_science)
17. Вимірюємо продуктивність за допомогою BenchmarkDotNet. — Режим доступу: https://habr.com/ru/post/277177/
18. WebSockets vs Polling? — Available from: https://stackoverflow.com/questions/44731313/at-what-point-are-websockets-less-efficient-than-polling
19. WebSockets vs. Server-Sent events — Available from: https://stackoverflow.com/questions/5195452/websockets-vs-server-sent-events-eventsource
20. WebSockets vs Long Polling — Available from: https://ably.com/blog/websockets-vs-long-polling
21. HTTP vs Websockets: A performance comparison — Available from: https://blog.feathersjs.com/http-vs-websockets-a-performance-comparison-da2533f13a77
22. K6 — Available from: https://k6.io/
23. Introduction to Scaleout in SignalR — Available from: https://docs.microsoft.com/en-us/aspnet/signalr/overview/performance/scaleout-in-signalr

24. Getting started with the Threat Modeling Tool — Available from: https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-getting-started
25. WebSocket security issues. — Available from: https://resources.infosecinstitute.com/topic/websocket-security-issues/
26. What is HTTP Vulnerability? — Available from: https://www.purevpn.com/ddos/http-vulnerability
27. WebSocket Security: Top 8 Vulnerabilities and How to Solve Them — Available from: https://www.neuralegion.com/blog/websocket-security-top-vulnerabilities/
28. WebSocket Security — Available from: https://devcenter.heroku.com/articles/websocket-security
29. WSS works on http? — Available from: https://stackoverflow.com/questions/34532006/wss-works-on-http/34554045#34554045
30. WS on HTTP vs WSS on HTTPS — Available from: https://stackoverflow.com/questions/26791107/ws-on-http-vs-wss-on-https
31. How to secure your WebSocket connections — Available from: https://www.freecodecamp.org/news/how-to-secure-your-websocket-connections-d0be0996c556/
32. Майбутнє Web це HTML через WebSockets. — Режим доступу: https://habr.com/ru/post/570522/
33. Fusion is a .NET library that implements DREAM – Distributed REActive Memoization. — Available from: https://www.reposhub.com/dotnet/distributed-computing/servicetitan-Stl-Fusion.html
34. Stl.Fusion — Available from: https://github.com/servicetitan/Stl.Fusion
35. Feathers — Available from: https://docs.feathersjs.com/