КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра прикладної статистики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 124 Системний аналіз

на тему:

СЕРВІС ДЛЯ ЗБЕРЕЖЕННЯ ТА УПРАВЛІННЯ ДАНИМИ

Виконав студент 4-го курсу

Роговий Семен Сергійович

_____

(підпис)

Науковий керівник:

доцент, доктор фіз.-мат. наук

Розора Ірина Василівна

_____

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри прикладної статистики

«5» червня 2023 р.,

протокол № 11

Завідувач кафедри

І. В. Розора _____

(підпис)

Київ – 2023

РЕФЕРАТ

Кваліфікаційна робота містить 66 сторінок, 2 таблиці, 14 рисунків, список використаних джерел з 26 найменувань.

Об’єктом дослідження є пошук і зберігання відповідей з цільових сторінок і, зокрема, з форм зворотного зв'язку.

Предметом дослідження в даній роботі є, безпосередньо, сам додаток. Він має поліпшити наявні функціональні можливості аналогічних рішень у цій галузі та включати всі функції, необхідні користувачам.

Метою роботи є створення веб-сервісу, що надає список предметів, яких бракує, і задовольняє вимоги користувачів. А також його практична реалізація, у вигляді справжнього сайту (інтернет-магазину).

Для досягнення мети необхідно виконати декілька кроків. По-перше, необхідно ретельно проаналізувати описувану цільову область. У результаті моделюється система і формалізуються вимоги. Для створення такого додатку розробникам необхідно знати мову програмування і бути знайомими з її функціями та інструментами, такими як обробка даних і тестування. Крім того, необхідно вжити заходів для розроблення масштабованої системи, яку можна буде легко розширювати та підтримувати в майбутньому.

За результатами дослідження було успішно розроблено та отримано повністю функціонуючий сервіс для збереження та управління даними. Цей сервіс включає функцію генерації форм та формування аналітики на основі наданих даних. Він повністю задовольняє вимоги та відповідає виявленим потребам, що були визначені під час аналізу існуючих рішень у даній предметній області.

Одержані результати можуть бути використані в практичній діяльності для роботи з користувачами.

Рік виконання кваліфікаційної роботи 2023.

Рік захисту роботи 2023.

Зміст

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CQRS (Command Query Responsibility Segregation) - це архітектурний підхід, який відокремлює операції запису (команди) від операцій запиту (запити).

СУБД – система управління базами даних.

REST (Representational State Transfer) - це архітектурний стиль для проектування розподілених систем, основаних на веб-протоколі HTTP.

Microservises (Мікросервіси) – Це набір невеликих модулів, на основі яких здійснюється безперервна поставка та розгортання великих і складних додатків.

SOA (Service Oriented Architecture) - Це архітектурний підхід, що базується на розробці програмного забезпечення як набору сервісів, які незалежно функціонують та взаємодіють між собою за допомогою стандартизованих протоколів.

API (Application Programming Interface) - це набір правил та протоколів, які визначають спосіб взаємодії між різними програмами.

ПЗ – програмне забезпечення.

CMS(Content Management System) − Система, яка використовується для забезпечення та організації спільного процесу створення, редагування та управління текстовими та мультимедійними документами.

CAPTCHA(Completely Automated Public Turing test to tell Computers and Humans Apart) − (капча) Повністю автоматизований публічний тест Тьюрінга для відрізнення комп'ютерів та людей.

DevOps(Development Operations) - Це ряд практик, що спрямовані на покращення взаємодії між розробниками та фахівцями з інформаційно-технологічного обслуговування, а також на зближення їхніх робочих процесів один з одним.

IDE (Integrated Development Environment) - це програмне забезпечення, яке надає розробникам зручне середовище для написання, відлагодження та тестування програмного коду.

SPA (Single-Page Application) - це тип веб-додатка, в якому весь необхідний HTML, CSS та JavaScript завантажується лише один раз при першому завантаженні сторінки, а подальша навігація та взаємодія з додатком відбувається без перезавантаження сторінки.

ВСТУП

Для багатьох галузей сфери послуг настав час переключити свою увагу на інтернет, який має великий потенціал для удосконалення та розвитку. Це пов'язано з тим, що зараз люди мають доступ до інтернету в будь-який час. Минуло багато часу відтоді, як людство стало повністю віртуальним, але це також дуже зручно, оскільки в кожного в кишені є засіб підключення до мережі.

Важливим кроком для компаній, які прагнуть отримати прибуток за допомогою інтернет-технологій, є перетворення відвідувачів сайту на потенційних клієнтів. Не бажаючи втрачати таку можливість, підприємці звертаються до одного з найефективніших інструментів онлайн-маркетингу: створення так званих цільових сторінок (Landing page). Це веб-сайти, що складаються, зазвичай, з однієї сторінки, яка спонукає користувачів вчинити певну дію, наприклад, купити товар або замовити послугу.

Landing page (Landing-сторінка) – важливий інструмент для просування бренду. Вони допомагають збільшити конверсію, залучити цільову аудиторію та вивести клієнтів на новий рівень. Для створення лендінгу не обов'язково наймати веб-дизайнера: ви можете скористатися відповідним конструктором сайтів [13].

Водночас постає запитання: "Як я можу підтримувати зв'язок зі своїми клієнтами?". Корисним інструментом для цього є форма зворотного зв'язку/контакту (feedback). Це один зі способів взаємодії клієнтів із менеджерами та власниками сайтів. Наприклад, з її допомогою можна отримати зворотний зв'язок від клієнтів, замовити послуги або залишити заявку, тощо.

Форми зворотного зв'язку вже давно стали найвизнанішим засобом спілкування між власниками ресурсів і користувачами. На це є багато причин.

По-перше, контактні форми значно спрощують надсилання листа, завдяки спеціальним полям для написання тексту. Такі шаблони роблять більш зручним замовлення певних послуг.

Ще однією перевагою є те, що користувачеві не потрібно вводити адресу електронної пошти власнику сайту, оскільки вона вже вказана у формі зворотного зв'язку. Крім того, адреса електронної пошти не показується відвідувачам сайту, що допомагає запобігти спаму та передачі власної інформації 3-ім особам. Зважаючи на величезну кількість переваг, які мають форми зворотного зв'язку, важко вигадати якісь недоліки, якщо такі взагалі існують. Крім того, що вони зручні для власника ресурсу і користувача, вони ще й економлять багато часу. Це пов'язано з тим, що люди, які хочуть щось дізнатись, не повинні витрачати дорогоцінний час на запуск свого поштового клієнта, щоб відправити листа. Все, що їм потрібно зробити, це перейти на сторінку з формою зворотного зв'язку і ввести своє ім'я, адресу електронної скриньки, необхідний текст і кнопку відправки в передбачені поля. У великих компаніях багато користувачів часто мають кілька різних електронних адрес, тому вони просто губляться у виборі одержувача. Знову ж таки, форма зворотного зв'язку може допомогти тут.

Відсутність зручного способу зберігання відгуків користувачів цільових сторінок (landing-сторінок), необхідність надання іншим розробникам і компаніям прикладів актуальності, подальшого розширення та потенціалу прибутку від розроблюваного додатка, а також автоматичної генерації форм зворотного зв'язку, здатних задовольнити потреби, що є присутніми в описаних галузях, визначає *актуальність* обраної теми кваліфікаційної роботи.

Об’єктом дослідження в даній роботі є пошук і зберігання відповідей з цільових сторінок і, зокрема, з форм зворотного зв'язку. Було оцінено кілька альтернативних рішень, доступних на ринку. Вони пропонують певні функції, але їх недостатньо для того, щоб повністю задовольнити потреби користувачів.

Предметом дослідження в даній роботі буде, безпосередньо, сам додаток. Він має поліпшити наявні функціональні можливості аналогічних рішень у цій галузі та включати всі функції, необхідні користувачам.

Метою роботи є створення веб-сервісу, що надає список предметів, яких бракує, і задовольняє вимоги користувачів. А також його практична реалізація, у вигляді справжнього сайту (інтернет-магазину).

Для досягнення цієї мети необхідно виконати декілька кроків. По-перше, необхідно ретельно проаналізувати описувану цільову область. У результаті моделюється система і формалізуються вимоги. Для створення такого додатку розробникам необхідно знати мову програмування і бути знайомими з її функціями та інструментами, такими як обробка даних і тестування. Крім того, необхідно вжити заходів для розроблення масштабованої системи, яку можна буде легко розширювати та підтримувати в майбутньому.

Кваліфікаційна робота складається з декількох частин (Переліку умовних позначень, вступу, чотирьох розділів, висновку та списку використаної літератури). Дослідження охоплюватиме такі етапи розробки, де ми почнемо з аналізу предметної області, а закінчимо представленням готового застосунку.

Перший розділ висвітлює плюси і мінуси таких додатків, аналізуючи існуючі рішення та їхні можливості.

Другий розділ відповідає за проектування майбутньої системи. Це основний етап розроблення застосунку, який містить у собі моделювання системи, визначення сутностей, що використовуватимуться в майбутньому, написання сценаріїв застосування, вибір архітектурних рішень, мов програмування та засобів розроблення.

У розділі під номером 3 описується етап розроблення системи та етап реалізації функціональності, а також використовувані технології.

У четвертому розділі розглянута практична реалізація розробленого веб-сервісу.

1 ОГЛЯД ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

У цьому розділі представлено огляд ринку існуючих доменів та їхніх альтернатив в інших сферах. Аналіз визначених сфер був проведений для того, щоб зрозуміти цілі розвитку, характеристики та активи, на які необхідно звернути увагу під час розробки.

У ньому також розглядається, які з функцій, перерахованих в аналогічних рішеннях, доцільно реалізувати в цій системі і чого не вистачає для повної реалізації потенціалу цієї галузі.

1.1 Огляд існуючих сервісів

На сьогоднішній час існує доволі мало сервісів, які можуть надати користувачам певний набір функцій, наприклад, зберігання даних цільової сторінки (landing-сторінки) або автоматичне створення форм зворотного зв'язку. Почнемо з, мабуть, одного з найпопулярніших сервісів для створення веб-форм. Звичайно це “Google Forms” (Рисунок 1.1). Це дуже проста і водночас досить гнучка платформа.

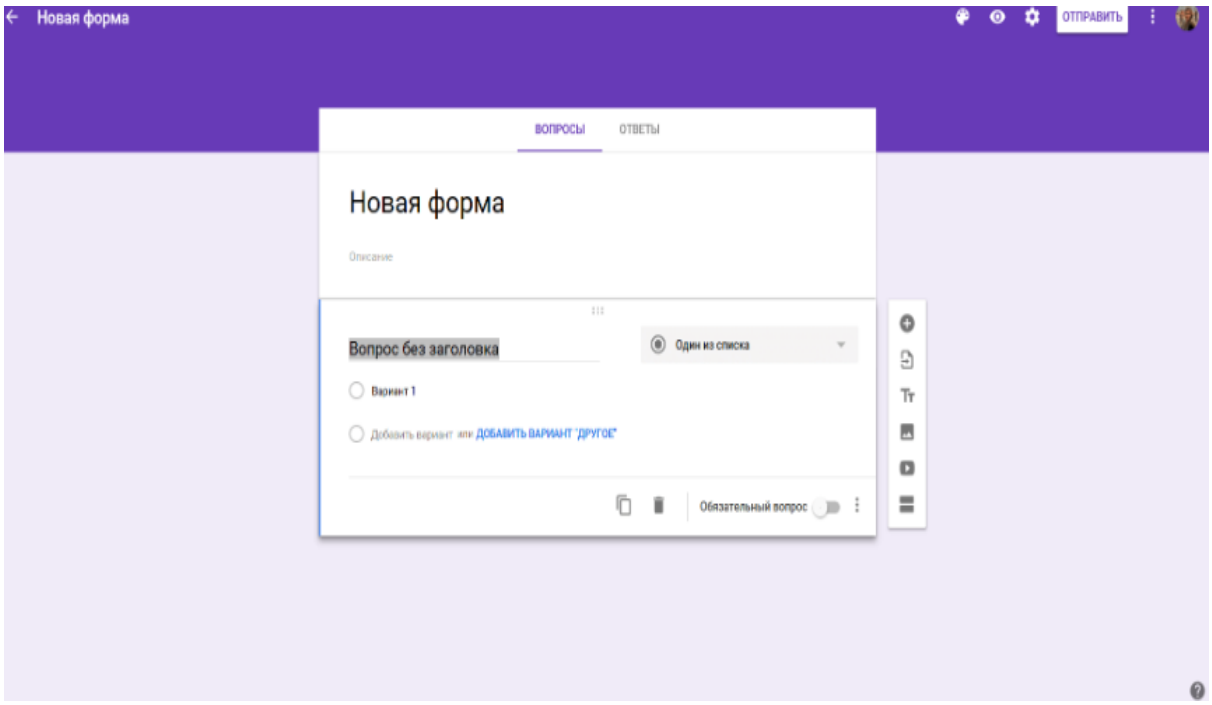

Рисунок 1.1- Вигляд Гугл Форми при роботі з нею

Окрім гнучкості, основними перевагами є обмежена кількість форм для заповнення та можливість налаштовувати сповіщення на електронну пошту.

Також зручність цього інструменту полягає в його:

-адаптивності - створення, редагування та перегляд матеріалів доступні на будь-яких пристроях;

-доступності - будучи в будь-якому місці і в будь-який час, користувач може працювати з додатком. Зберігаються Google Forms для опитування на Google Диску;

-легкості використання - полягає в зручності роботи на всіх етапах, від створення до заповнення респондентами;

-унікальному оформленні - для дизайну можна вибрати готові теми Google Forms або додати власні зображення;

-зручності аналізу - після заповнення форми інформація автоматично обробляється додатком, і ви зможете отримати готову статистику відповідей[12].

З іншого боку, можливості кастомізації та кількість шаблонів (17 типів) вкрай обмежені, а інтеграція посередині цільової сторінки неможлива. Ці недоліки забили досить великий цвях у труну користувацького досвіду.

Перейдемо до сервісу під назвою «FormDesigner»(Рисунок 1.2) у котрого є можливість створювати динамічні форми .У ньому ,крім динамічних форм, ви можете створювати власні веб-форми з різними темами дизайну та контентом[14].

З огляду на перераховані можливості, в додатку не вистачає деяких функцій, які ми хотіли б мати. Наприклад, контактна форма не має з'єднання з віддаленим сервером для зберігання залишеного запиту. Це значно б спростило процес обробки інформації власнику цільової сторінки, на якій розташована форма. Ймовірно це пов’язано з тим, що в першу чергу розробники спрямували всі свої сили та енергію на реалізацію якісного та швидкого конструювання різних форм та їх модифікацію.

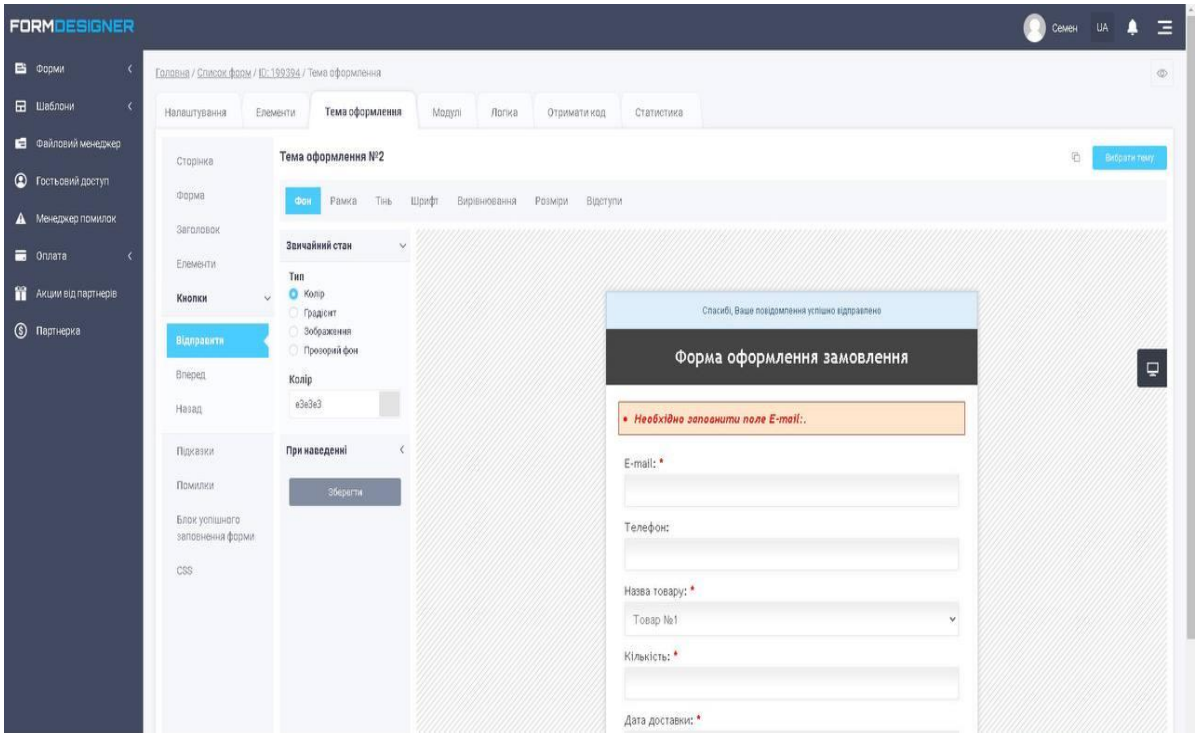

Рисунок 1.2 – Вигляд FormDesigner при створенні форм

Серед інших особливостей FormDesigner це те що, створені за його допомогою веб-форми працюють на будь-яких сайтах та CMS. При необхідності їх можна захистити паролем або за допомогою CAPTCHA. А якщо в веб-формі використовується велика кількість елементів, їх можна розділити на декілька сторінок, що спрощує заповнення форми клієнтами.

Також можемо звернути свою увагу на конструктор веб-форм під назвою “QuintaDB”(Рисунок 1.3). QuintaDB - це онлайн-конструктор веб-форм та баз даних. Ми можемо створювати та налаштовувати форми, діаграми, карти всього лише за кілька клацань. Є можливість зворотного зв’язку, і ми можемо отримувати повідомлення по електронній пошті та SMS.Також доступні такі можливості, як задавати умовні дії та умовне форматування, наявність клієнтського порталу. Також є функція зберігання даних онлайн у реляційній базі даних з зручним і ефективним інтерфейсом, без необхідності купувати власний сервер. Доступ до інформації 24 години на добу з будь-якого місця

світу[15]. Серед мінусів, можна виділити лише несуттєве – доволі тяжкий інтерфейс.

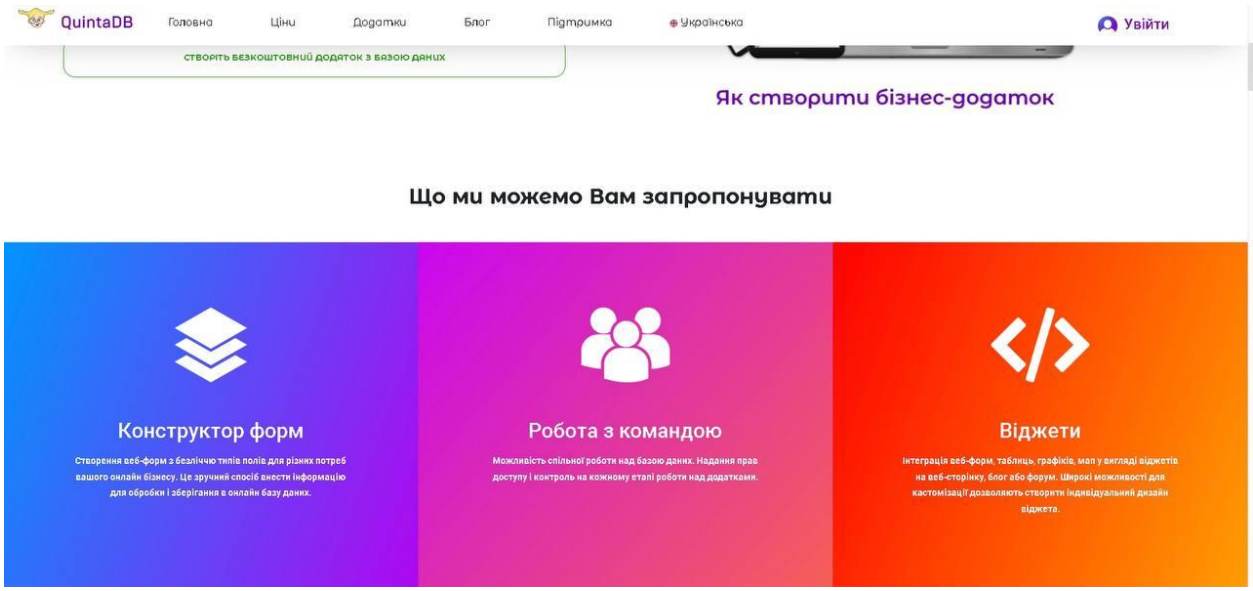

Рисунок 1.3 – Вигляд QuintaDB

Існує багато інших застосунків, але ті, що розглянуті в цьому розділі, ілюструють основні переваги функціональності, яку пропонують конструктори веб-форм для цільових сторінок. Однак хотілось би мати один такий додаток, який би міг зібрати у себі , якщо не всі, то ключові особливості, характерні для цього сектору.

Прикладом функцій такого додатку, наприклад, мають бути забезпечення повністю налагодженої форми зворотного зв'язку, яка гармонійно вписувалась б у стиль будь-якого веб-сайту і мала розширений функціонал, такий як можливість прикріплювати додаткові файли. Окрім текстової інформації, потрібно мати можливість зберігати отримані заявки на віддаленому сервері і легко отримувати до них доступ. Крім того, інтеграція цього додатка на лендінг-сторінку повинна бути дуже простою і не вимагати спеціальних знань.

Отже, для створення сервісу збереження і керування даними потрібно поєднати найкращі характеристики наявних рішень, забезпечивши при цьому достатню гнучкість в налаштуванні, але зберігаючи максимальну простоту використання, щоб не відлякувати клієнтів. Зрозуміло, що безпека даних і захист від спаму також є важливими аспектами, про які не можна забувати.

Важливо врахувати, що не всі функції можна реалізувати через їх доцільність, оскільки система повинна мати консистентну структуру і не розсіювати увагу на маловажливі деталі, які не відповідають її основному призначенню.

В таблиці 1 представлена порівняльна характеристика аналізованих додатків, включно з розроблювальним сервісом, який демонструє очевидні переваги над конкурентними рішеннями в даній галузі. Крім того, таблиця включає можливості, які можуть бути подібними або навпаки.

Таблиця 1 – Таблиця порівняння для існуючих сервісів.

Пункти порівняння	Google Forms	FormDesigner	QuintaDB	Розроблювальним сервіс
Створення форми	√	√	√	√
Можливість поєднання	–	√	√	√
Можливості індивідуалізації	–	√	–	√
Гнучкість конструювання	√	–	√	√
Огляд вхідних даних	√	–	√	√
Обробка вхідних даних	√	–	√	√
Індивідуальні пропозиції	√	–	–	√

1.2 Склад предметної області

Для кращого розуміння предметної області та ефективного проектування системи необхідно спочатку проаналізувати її складові елементи та взаємодію між ними. Перший крок у цьому процесі - виокремлення сутностей, які відіграють роль у процесах, пов'язаних з даною областю, а також визначення додатків, які забезпечують необхідний функціонал.

Серед сутностей, які можна виділити, у даній області є наступні:

1. Форма зворотного зв'язку: Це одна з ключових сутностей, яка використовується для отримання даних. Вона містить інформацію, яку користувачі надають через форму, і дозволяє здійснити взаємодію з системою.

2. Таблиця з отриманими даними: Ця сутність включає набір даних, які були отримані з лендінг-сторінки. Вона служить для зберігання та організації цих даних.

3. Лендінг-сторінка: Це веб-сторінка, яка інтегрує форму зворотного зв'язку. Лендінг-сторінка зазвичай використовується для залучення потенційних користувачів та надання їм інформації про продукт або послугу.

4. Клієнт сервісу: Ця сутність представляє собою особу або організацію, яка є власником лендінг-сторінки і використовує систему. Клієнт сервісу є потенційним користувачем системи і може виконувати різні дії, пов'язані зі збором та обробкою отриманих даних.

Описані сутності грають важливу роль у взаємодії та обміні інформацією у цій предметній області.

Після виокремлення сутностей важливим кроком в описі структури предметної області є розгляд взаємодії між цими сутностями та процесами, у яких вони беруть участь. Це дозволяє визначити необхідний функціонал системи та включити його до проектування.

Враховуючи специфіку розробки програмного забезпечення і встановлені стандарти оформлення коду, наведені сутності будуть використовуватися у подальшому описі проектування та розробки системи, але можуть мати інші

назви та формулювання, що відповідають вимогам програмування. Вони продовжать виконувати ключову роль у функціонуванні системи, але будуть представлені у вигляді компонентів, класів або об'єктів, що відповідають стандартам розробки програмного забезпечення.

Далі в аналізі предметної області розглядається взаємодія між сутностями та процеси, в яких вони беруть участь, і визначається, які аспекти цієї взаємодії слід включити до функціоналу системи.

Для зручності можна виділити основні можливості, які пропонує дана сфера, але описати лише ті, які відсутні у наявних рішеннях і які можна реалізувати програмно. Зокрема, можна зосередитися на таких варіантах використання:

1. Створення форми та вибір даних: Користувач сервісу має можливість створювати форму зворотного зв'язку і визначати, які дані будуть зберігатися в таблиці. Це дає змогу користувачам налаштовувати форму під свої потреби та збирати конкретну інформацію, яка їм потрібна.

2. Перегляд статистики отриманих даних: Користувачі сервісу мають можливість переглядати статистику та аналізувати отримані дані. Це дозволяє їм отримувати уявлення про кількість та характеристики зібраних даних, що може бути корисним для прийняття рішень або виявлення трендів.

3. Обробка отриманих даних: Користувачі сервісу можуть обробляти отримані дані, наприклад, надсилати електронні листи на електронну пошту користувачів, які використали форму зворотного зв'язку на лендінг-сторінці. Це дає можливість здійснювати подальшу комунікацію зі зібраними даними та взаємодію з потенційними клієнтами.

Описані вище можливості є унікальними та відсутніми у наявних рішеннях, тому їх реалізація є доцільною та можливою з програмної точки зору.

Зазначені сценарії показують, що більшість дій у розроблюваній системі відбуваються за ініціативою користувача, і спрямовані на його користь. Це дозволяє виділити користувача як головну активну особу системи, в якій використання інших об'єктів випливає з його потреб.

2 КОНСТРУЮВАННЯ СИСТЕМИ

В даному розділі будуть описані послідовні етапи розробки сервісу. Ми розглянемо впровадження різних етапів проектування та проаналізуємо нюанси та обмеження, які необхідно враховувати, оскільки вони мають вплив на кінцевий результат.

У подальших пунктах буде приділено особливу увагу кожному етапу розробки, а також для кращого розуміння будуть надані таблиці та схеми.

2.1 Системне моделювання

За допомогою кількох прикладів ми показуємо, як можна моделювати систему та аспекти її проектування. Основний акцент тут зроблений на продуктивності, оскільки продуктивність є найбільш яскравим прикладом цінності моделювання і чудовим ілюстрацією підходу до моделювання[16].

При розробці розподілених інформаційних систем та алгоритмів та програмного забезпечення, що забезпечують їх функціонування, широко використовується метод імітаційного моделювання. На сьогоднішній день існує значна кількість спеціалізованих програмних систем, які дозволяють будувати імітаційні моделі розподілених систем і проводити імітаційні експерименти, збирати статистичні дані та представляти результати імітаційного моделювання.

У процесі моделювання першочерговим завданням є формування бізнес-вимог. У великих компаніях це завдання зазвичай виконують бізнес-аналітики, які спілкуються з клієнтами і розуміють їх запити щодо кінцевого результату.

В даному випадку формування вимог до системи відбулося під час огляду та аналізу предметної області. Перш ніж перейти до моделювання, необхідно зрозуміти, що модель є абстракцією з області програмування, яка складається з сутностей, що існують у предметній області.

Після виділення всіх основних сутностей, які виступають ключовими елементами в описі області, потрібно переходити до інших етапів проектування. Крім того, необхідно розробити алгоритми для ефективного впровадження цих сутностей в систему зберігання та управління даними.

2.1.1 Сутності системи

При проектуванні системи одним з методів моделювання предметної галузі є використання моделі "сутність-зв'язок". У цій моделі основними поняттями є сутність, атрибут і зв'язок.

Сутність представляє об'єкт реального світу, який має інстанси або екземпляри, що відрізняються один від одного за значеннями їх атрибутів. Атрибут визначає властивість або характеристику сутності. Зв'язок, з свого боку, визначає взаємодію між різними сутностями.

Спочатку необхідно ідентифікувати всі необхідні сутності, які будуть використовуватися в розробці системи і яких буде достатньо для опису обраної предметної області.

Ці сутності будуть представлені у вигляді таблиці, в якій будуть вказані назва сутності, її взаємодія з іншими сутностями та функції, які вона виконує всередині системи[17].

Таблиця 2 – Характеристика сутностей системи

Сутність	Зв’язок	Атрибут
User(користувач)	Форма Аналітика Набір даних	Він має можливість створювати нові форми, переглядати отримані дані та обробляти їх. Крім того, також є можливість перегляду аналітики на основі цих даних.

Dataset(набір даних)	Користувач Форма	Включає в себе інформацію про поля форми та зберігає отримані дані.
Form(форма)	Користувач	Основна функція створення форми та надання готового коду для її інтеграції на лендінг-сторінку.
Analytics (аналітика)	Набір даних	Включає в себе аналітичні дані, що базуються на збережених даних у наборі даних користувача.

Описані сутності відображають вимоги до розроблюваного додатку та показують, які переваги отримають користувачі від цього продукту.

Аналізуючи отримані результати, можна бачити, що користувач формує потрібний для нього набір даних, з якого сервіс налаштовує форму. Аналітична сутність в свою чергу аналізує ці дані у наборі.

Цей функціонал має одну основну мету - спростити дії, які користувач повинен виконати, щоб досягти своїх цілей, а саме: створити власну форму, зберігати інформацію, обробляти та організовувати її за своїми потребами тощо.

Під час безпосередньої розробки веб-сервісу варто дотримуватися цих встановлених вимог, щоб досягти бажаного результату.

2.1.2 Сценарії взаємодії з системою

Use case (використання випадку) - це докладний опис сценаріїв або ситуацій, в яких продукт, система або послуга може бути використана. Він визначає, яким чином виробничі можливості будуть застосовані для вирішення конкретних проблем або задоволення потреб користувачів.

Use case може бути представлений у формі текстового опису, діаграми або сценарію. Він зазвичай включає акторів (користувачів або системи, які

взаємодіють з продуктом), передумови (початкові умови), дії, які виконує продукт або система, та очікувані результати.

Use case допомагає розробникам, бізнес-аналітикам та іншим зацікавленим сторонам зрозуміти, як продукт або система взаємодіє з користувачами або іншими системами у реальних умовах. Він дозволяє виявити потенційні проблеми, визначити вимоги до функціональності та забезпечити належне розуміння того, яким чином продукт буде використовуватися у реальному середовищі.

Use case також використовується в процесі тестування продукту або системи для визначення тестових сценаріїв, які охоплюють різні можливості використання та перевіряють, чи працює продукт або система правильно у всіх передбачуваних умовах.

Сценарій використання є методологією, що застосовується в системному аналізі для визначення, пояснення та організації системних вимог. Він складається з набору можливих послідовностей взаємодій між системами та користувачами в певному середовищі, спрямованих на досягнення певної мети. Основна ідея цієї методології полягає в створенні документа, який описує всі кроки, які користувач виконує для завершення певної дії.

Тестування за допомогою юзкейсів виконується для виявлення додаткових логічних прогалин та помилок у веб-додатку, які можуть бути складними для виявлення під час тестування окремих модулів або частин додатку.

Зазвичай, сценарій використання описує, що система робить, а не як саме це відбувається. Це правило варто дотримуватися при створенні таких сценаріїв.

За допомогою юзкейсів можна описувати взаємодію між двома або більше учасниками, які мають конкретну мету. Наприклад:

- Покупка товару в магазині (покупець - продавець).
- Відправка листа електронною поштою (адресант - поштовий клієнт).
- Запит сторінки браузером (браузер - веб-сервер)[18].

Сценарії використання описують функціональні вимоги системи з перспективи кінцевого користувача, створюючи послідовність подій, яка спрямована на досягнення конкретної цілі. Вони полегшують сприйняття для користувачів і розробників. Повний набір сценаріїв використання може включати лише один основний шлях або основний шлях разом з різними альтернативними шляхами. Альтернативний шлях, відомий також як розширений сценарій використання, описує варіації основного шляху та незвичайні ситуації.

Сценарій використання відображає такі характеристики:

- Організацію функціональних вимог.
- Моделювання цілей взаємодії між компонентами.
- Відображення шляхів від подій до досягнення цілей.
- Опис одного основного сценарію подій та різних альтернативних сценаріїв.

Сценарії використання в бізнесі — це абстрактний опис, який не залежить від конкретних технологій і фокусується на бізнес-процесі та учасниках цього процесу. Ці сценарії визначають послідовність дій, які бізнес повинен здійснити для досягнення помітного результату для кінцевого користувача.

З іншого боку, сценарії використання системи надають більш детальний опис, посилаючись на конкретні процеси, які відбуваються у різних компонентах системи для досягнення цілей користувача. Діаграми варіантів використання системи детально описують функціональні вимоги, включаючи залежності та внутрішні допоміжні функції.

Єдині сценарії використання мають кілька переваг для розробників, оскільки вони вказують, як система повинна вести себе і допомагають виявити можливі помилки.

Така методологія має інші переваги:

- Список цілей, створений при написанні сценаріїв використання, дозволяє визначити складність системи.

- Фокусуючись на користувачі і системі, можна заздалегідь визначити реальні системні потреби ще на етапі проектування.

- Текстовий формат сценаріїв використання дозволяє легко зрозуміти їх зацікавленим сторонам, таким як клієнти, користувачі і керівники, а не тільки розробникам і тестувальникам.

- Створення розширених варіантів використання та визначення винятків з успішних сценаріїв допомагає економити час розробників і спрощує встановлення системних вимог.

- Зосередження на тому, що система повинна робити, а не як вона має це робити, допомагає уникнути передчасного проектування.

Крім того, сценарії використання можна легко перетворити на тестові випадки, збираючи дані для кожного з них. Ці тестові приклади допомагають розробникам переконатися, що всі функціональні вимоги системи включені в план тестування.

Варіанти використання також можуть бути корисними в інших аспектах розробки програмного забезпечення, таких як планування проекту, документація користувача та визначення тестових випадків. Сценарії використання можуть служити інструментом планування для ітеративної розробки.

Давайте розглянемо приклад використання (Use Case) для оформлення лікарняного листка співробітником через чат-бота. Сценарії роботи можуть включати в себе наступні елементи (причому кількість цих елементів залежить від складності сценарію):

Система: Внутрішній корпоративний чат-бот

Дійова особа: Співробітник компанії

Мета: Оформлення лікарняного

Тригер: Співробітник вирішує оформити лікарняний і відкриває діалог з чат-ботом

Результат 1: Інформація про дату лікарняного збережена

Результат 2: Співробітник успішно оформив лікарняний

За цією схемою, співробітник компанії веде діалог з чат-ботом, використовуючи його для оформлення лікарняного. Це дозволяє співробітнику досягти своєї мети.

Основний потік дій Use Case описує послідовність подій, яка веде до успішного досягнення мети співробітника.

Тригер є подією, що ініціює початок сценарію. В цьому випадку, тригером може бути рішення співробітника оформити лікарняний і почати діалог з чат-ботом.

Результат є досягненням або станом, який залишається після завершення сценарію. В даному випадку, результатом є збережена інформація про дату лікарняного та успішне оформлення лікарняного співробітником.

Цей Use Case використовується для проєктування можливих рішень та може бути використаний командою розробки та технічної підтримки чат-бота у їхній роботі.

Use Case виявляє, як система повинна діяти в різних ситуаціях і як задовольняти запити користувачів.

Використання Use Case у процесі розробки залежить від використовуваної методології. У деяких методологіях достатньо короткого огляду Use Case, тоді як інші методології дозволяють більшу гнучкість і зміни сценаріїв використання протягом розробки.

У деяких методологіях Use Case можуть починатися як короткі бізнес-сценарії, а потім розширюватися до детальних системних сценаріїв використання, а надалі перетворюватися на докладні тести, що охоплюють багато деталей.

Загалом, використання Use Case може варіюватися в залежності від методології розробки та потреб проекту.

Процес написання Use Case включає ідентифікацію всіх користувачів системи та створення профілю для кожного з них. Це означає визначення ролей, які вони виконують при взаємодії з системою. Для кожного користувача обирається мета або те, що вони намагаються досягти, використовуючи

систему. Кожна з цих мет стає окремим варіантом використання. Для кожного варіанту використання системи описується послідовність подій, яка призводить до досягнення цієї мети. Загальні елементи, які з'єднуються в сценаріях, дозволяють створити загальні випадки використання і написати опис для кожного з них.

В подальшому, наведу приклади 1,2 та 3 , які ілюструють ключові сценарії роботи системи і повністю відображають основні вимоги до неї.

Приклад 1 – Сценарій налаштування нової форми

Назва сценарію: Налаштування нової форми

Опис: Користувач має можливість створювати нові форми згідно своїх потреб і вимог.

Вхідні дані: Налаштування полів форми та персоналізаційні дані.

Алгоритм:

1. Користувач конфігурує форму шляхом визначення необхідних полів та їх типів даних.
2. Користувач вносить зміни до дизайну форми відповідно до своїх потреб.
3. Система генерує відповідний код та надає інструкції для інтеграції форми на лендінг-сторінку.

Результат: Користувач отримує згенерований код та інструкцію для успішної інтеграції нової форми на лендінг-сторінку.

Приклад 2 – Сценарій для огляду даних, що були отримані

Назва сценарію: Перегляд даних, які були отримані

Опис: Користувач бажає переглянути дані, які були отримані з landing-сторінки.

Вхідні дані: Таблиця з даними, яку користувач бажає переглянути.

Алгоритм: Користувач вибирає необхідну таблицю, пов'язану зі сторінкою, яка його цікавить.

Результат роботи: Користувач бачить на екрані потрібні дані і має можливість переглянути та обробити їх, або завантажити у відповідному форматі.
Ця інформація дозволить користувачеві зручно працювати з отриманими даними та виконувати необхідні операції над ними.

Приклад 3 – Сценарій перегляду проаналізованих даних по будь-якій обраній таблиці
Назва сценарію: Огляд проаналізованих даних
Опис: Користувач може переглядати автоматично згенеровану аналітику, що базується на вибраній таблиці з даними.
Вхідні дані: Обрана таблиця для аналізу з даними.
Алгоритм: Користувач обирає необхідну таблицю, яка прив'язана до певної сторінки, та активує відображення аналітики.
Результат роботи: Користувачу відображаються аналітичні дані, що автоматично створені на основі вибраної таблиці. Користувач має можливість переглянути та обробити ці дані, а також завантажити їх у потрібному форматі.

Після проведення аналізу предметної області та ідентифікації основних елементів системи було виявлено можливість створення сценаріїв роботи. Для ефективного відображення та максимізації користі від цих сценаріїв було вирішено використовувати діаграми станів та послідовності.

2.2 Побудова формальної системи

Враховуючи потреби предметної області і складність системи, деякі аспекти, які входять до цього етапу, можуть не бути доцільними або бути відкладеними на пізніші стадії розробки. Такі рішення можуть бути прийняті з метою оптимізації процесу або через недостатню вагомість цих аспектів у даному контексті. В деяких випадках, окремі етапи можуть бути об'єднані або навіть пропущені на етапі формалізації, і розглянуті напряму під час

написання коду або ігноровані взагалі, залежно від потреб проекту та обставин.

Після процесу формалізації отримується чітка і структурована модель, яка включає опис системи з визначеними межами. Цей опис може бути представлений різноманітними діаграмами та графіками, які демонструють компоненти системи, їх функції або алгоритми виконання операцій. Така модель надає візуальне відображення системи і дозволяє чітко розуміти її структуру та функціональні можливості.

Основні кроки формалізації, які використовуються в цьому дослідженні, включають:

1. Встановлення загальної структури системи і взаємозв'язків між її компонентами.
2. Визначення мови програмування та підходів до розробки програмного забезпечення, що найкраще пі дходять для вирішення поставленої задачі.
3. Використання абстракцій мови програмування та шаблонів проектування для спрощення розробки, забезпечення можливості розширення системи та покращення якості коду.
4. Використання готових фреймворків та бібліотек для реалізації конкретних функціональних можливостей системи та прискорення процесу розробки.
5. Створення необхідної інфраструктури, такої як сервери, бази даних, мережі, забезпечення безпеки і т.д., для ефективного функціонування системи.

Для забезпечення правильної розробки системи, яка в майбутньому може зазнати змін або розширень, і при цьому залишитися стійкою, цей перелік етапів формалізації є необхідним мінімумом.

Слід наголосити, що внесення змін у один аспект системи не повинно впливати на інші аспекти. Наприклад, зміна мови програмування не повинна вимагати зміни архітектури або бази даних. Кожен компонент системи повинен бути незалежним і може бути змінений окремо від інших з метою відповіді на нові вимоги чи потреби. Забезпечення такої незалежності дозволяє системі гнучко адаптуватися до змін і сприяє збереженню стійкості та ефективності.

2.2.1 Загальна структура та архітектура

У сфері програмування існує широкий спектр архітектурних рішень, кожне з яких має свої особливості та застосування в певних областях. Неможливо виокремити єдину архітектуру, яка була б універсальною і не залежала б від вимог і контексту розробки. Кожен стиль архітектури унікальний і має свої переваги та недоліки.

Під час оцінки різних архітектурних підходів і врахування потреб предметної області, було прийнято рішення використовувати мікросервісну архітектуру. Цей вибір був зроблений після ретельного аналізу та спроби відповідати потребам конкретної системи(Рисунок 2.4).

Мікросервісна архітектура (або просто "мікросервіси") - це підхід до створення додатків у вигляді набору незалежних, самостійно розгортаних сервісів, які є децентралізованими і розробляються незалежно один від одного. Ці сервіси мають слабку зв'язність, можуть бути незалежно розгорнуті і легко обслуговуються. Монолітний додаток створюється як єдине недільне ціле, у той час як у мікросервісній архітектурі його розбивають на безліч незалежних модулів, кожен з яких вносить свій внесок у загальну справу. Мікросервіси нерозривно пов'язані з DevOps, оскільки є основою методики безперервної доставки, завдяки якій команди можуть швидко адаптуватися до потреб користувачів.

Мікросервіс - це веб-сервіс, відповідальний за один елемент логіки в певній предметній області. Додаток створюється як комбінація мікросервісів, кожен з яких надає функціональні можливості у своїй предметній області. Мікросервіси взаємодіють один з одним через API-інтерфейси, такі як REST або gRPC, але не мають інформації про внутрішню структуру інших сервісів. Таке згуртоване взаємодія між мікросервісами називається мікросервісною архітектурою[19].

Особливість мікросервісної архітектури полягає у поділі системи на невеликі компоненти, кожен з яких працює у власному програмному середовищі і виконує конкретну функціональність. Ці невеликі компоненти можуть бути розгорнуті окремо на різних серверах і машинах, за допомогою спеціальних інструментів, призначених для цього [1].

Функціонал мікросервісів визначається шляхом аналізу бізнес-потреб, які встановлюються під час комунікації з замовником або під час моделювання системи, коли розробляються сценарії роботи.

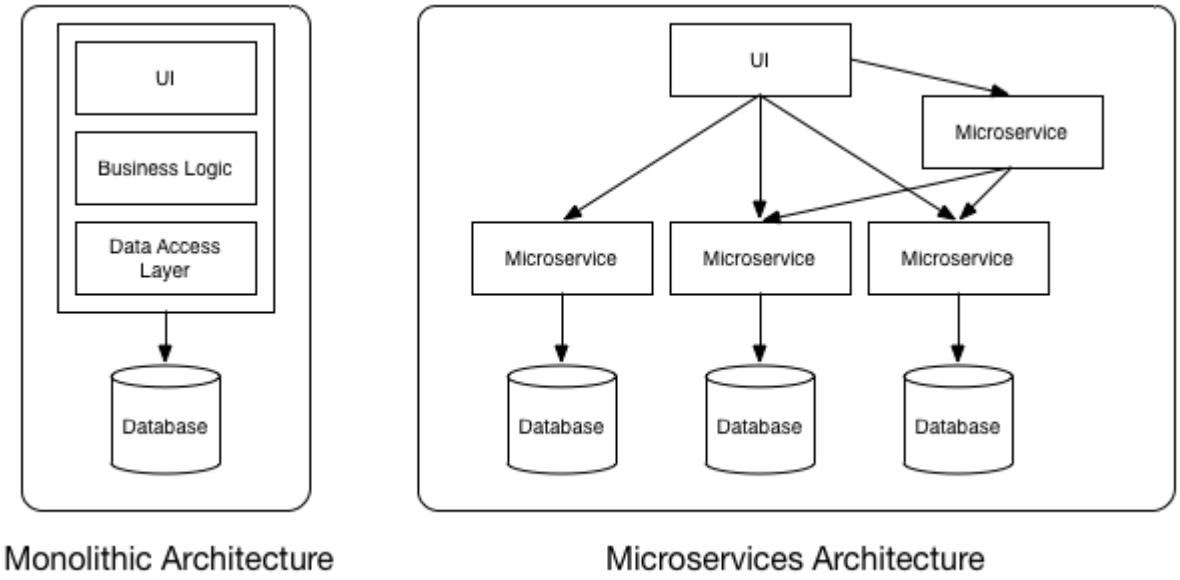

Рисунок 2.4 - Структура мікросервісної архітектури [8].

Мікросервісна архітектура, або просто мікросервіси, є способом розробки програмних додатків, де система розбивається на набір невеликих модульних сервісів, що можуть самостійно розгортатися. Кожен мікросервіс виконує свій унікальний процес.

Мікросервісна архітектура є масштабованою архітектурою, яка дозволяє підтримувати різноманітні платформи і пристрої. Вона відмінна від монолітної архітектури, де весь додаток побудований як єдиний автономний блок. У монолітній архітектурі серверна програма виконує обробку HTTP-

запитів, логіку та взаємодіє з базою даних. Навіть невелика зміна може вимагати створення та розгортання нової версії всього додатку.

Однією з ключових відмінностей в інфраструктурі системи є те, що кожен сервіс має власну окрему базу даних, до якої має прямий доступ лише сам сервіс. Це зроблено для забезпечення цілісності даних та уникнення можливості їх неправильного використання або витоку.

У випадку потреби отримати доступ до даних, що зберігаються в базі одного з сервісів, можна скористатися набором інструментів Системи управління базами даних (СУБД) для створення "матеріалізованих таблиць". Ці таблиці призначені виключно для читання і містять готові дані, що можуть бути використані безпосередньо.

Для забезпечення взаємодії між сервісами використовуються різноманітні сторонні інструменти та спеціальні протоколи обміну даними. Серед найбільш популярних можна виділити наступні:

- Secure Shell (SSH) - це протокол рівня додатків, який використовується для безпечного віддаленого керування системою через захищений канал. Цей протокол широко використовується в різних технологіях. Підключення за допомогою SSH мають наступні особливості:

    1. Шифрування: SSH використовує авторизацію за ключем, що дозволяє шифрувати весь трафік, включаючи паролі, за допомогою різних алгоритмів.

    2. Безпека: шифрування допомагає забезпечити безпеку під час віддаленої роботи, підвищуючи надійність з'єднання.

    3. Можливість стиснення: SSH підтримує можливість стиснення інформації під час передачі.

    Копіювання файлів через SSH дозволяє підвищити рівень захисту під час передачі даних. SSH вважається протоколом на рівні додатків і призначений для забезпечення безпечного віддаленого доступу.;

- Hyper Text Transfer Protocol (HTTP) є основним протоколом для функціонування всіх інтернет-ресурсів. Його головне завдання - надання можливості запитувати ресурси, які потрібні у віддаленій системі, наприклад, файлів.

  HTTP працює за принципом клієнт-серверної взаємодії, де одна сторона надсилає дані, а інша сторона їх обробляє. Особливості протоколу HTTP пов'язані з його простотою, можливістю розширення, сесійністю та залежністю від транспортного рівня.

  Протокол HTTP відомий своєю простотою, що дозволяє легко розуміти алгоритми його роботи. Він також може бути розширений шляхом узгодження між клієнтами і серверами, що дозволяє змінювати його семантику.

  HTTP є сесійним протоколом, де кожен запит вважається окремою сесією і може ділитися певним контекстом, але взаємозв'язок між запитами відсутній.

  Протокол HTTP працює на рівні транспорту, використовуючи управління на транспортному рівні для забезпечення передачі даних[20].

2.2.1.1 Вимоги до користування

Для визначення найбільш доцільного архітектурного підходу до проектування необхідно аналізувати вимоги, які були ідентифіковані під час розгляду альтернативних рішень в предметній області, а також розробити сценарії використання. Цей аналіз допоможе зрозуміти, які основні вимоги необхідно врахувати при розробці архітектури.

Зокрема, можна виділити наступні ключові вимоги до архітектури:

- Масштабованість: Архітектура повинна бути масштабованою, що дозволить легко збільшувати ємність системи за потреби;
- Швидкодія: Архітектура повинна бути ефективною та швидкою, забезпечувати оптимальне використання ресурсів та мінімальні затримки в роботі програми;
- Безпека: Архітектура повинна забезпечувати високий рівень безпеки, включаючи захист від зловмисних атак, збереження конфіденційності даних та забезпечення цілісності і доступності системи;
- Прозорість. З урахуванням масштабування проекту і включення нових розробників у майбутньому, важливо, щоб код був легким для читання та розуміння. Це допоможе новим людям швидко освоїти проект та внести свій внесок у розробку.

Враховуючи вищевказані вимоги до розроблюваної системи, можна прийти до висновку, що найбільш підходящим архітектурним підходом буде мікросервісна архітектура, яка забезпечить задоволення всіх цих вимог. Особливістю цього підходу є можливість легкого моніторингу та збору статистики, що допоможе виявити слабкі місця в системі та покращити їх. Таким чином, замість повного розгортання системи заново, можна просто замінити або додати окремий сервіс для виправлення проблеми. Мікросервісний підхід є популярним серед великих компаній, оскільки він демонструє свої переваги у масштабованих високонавантажених системах[2].

2.2.1.2 Плюси та мінуси обраного рішення

Щоб пояснити вибір даного рішення та порівняти його з іншими підходами, треба розглянути доцільність та недоцільність мікросервісної архітектури. Це допоможе зрозуміти причину обрання даного рішення та провести об'єктивне порівняння.

Переваги мікросервісної архітектури включають:

- Масштабованість: Мікросервіси дозволяють гнучко масштабувати окремі компоненти системи залежно від потреб, що сприяє ефективному використанню ресурсів та забезпечує високу продуктивність.

- Незалежність: Кожен мікросервіс може бути розроблений, розгорнутий та масштабований незалежно від інших компонентів системи, що дозволяє швидко розвивати та вдосконалювати окремі частини програмного забезпечення.

- Гнучкість: Мікросервіси дають змогу використовувати різні технології та мови програмування для кожного компонента системи в залежності від його потреб, що сприяє вибору найкращих інструментів для конкретних завдань.

- Простота супроводу: Кожен мікросервіс може бути розгорнутий, оновлений та масштабований окремо, що полегшує супровід та виправлення помилок в системі.

- Командна робота: Мікросервісна архітектура дає змогу розробляти окремі сервіси незалежно один від одного, що сприяє паралельній роботі розробників та підвищує продуктивність розробки.

Недоліки мікросервісної архітектури включають:

- Складність: Керування та координація між багатьма мікросервісами може бути складним завданням, особливо при великому обсязі мікросервісів, що може вимагати додаткових зусиль для забезпечення їх взаємодії та синхронізації.

- Системна складність: Використання мікросервісної архітектури може призвести до збільшення загальної складності системи, оскільки потрібно керувати багатьма окремими сервісами з їх власними базами даних, комунікаційними протоколами та інфраструктурою.

- Мережева навантаженість: Мікросервіси взаємодіють через мережу, що може призвести до збільшення мережевого навантаження та затримок при обміні даними між сервісами.

- Потреба у додаткових ресурсах: Для розгортання та управління мікросервісами можуть знадобитися додаткові ресурси, такі як сервери, інструменти автоматизації та моніторингу, що може призвести до збільшення витрат.

- Складність тестування: Тестування мікросервісної системи може бути складним завданням через необхідність валідації взаємодії між різними сервісами та розгорнутими екземплярами кожного сервісу.

Враховуючи ці переваги та недоліки, вибір мікросервісної архітектури був зроблений з огляду на потреби проекту та його майбутню масштабованість. Хоча мікросервіси можуть вимагати додаткових зусиль у керуванні та координації, вони надають гнучкість, незалежність та простоту супроводу, що є важливими перевагами для масштабованих систем[21].

Розглянемо альтернативи мікросервісній архітектурі. Альтернативи мікросервісної архітектури включають:

1. Монолітна архітектура: В цьому підході всі компоненти системи розгорнуті разом у єдиному додатку. Він зручний для невеликих проектів або на початкових етапах розробки, оскільки не потребує складності управління між багатьма сервісами. Однак, монолітна архітектура може стати незручною при масштабуванні та розвитку системи, оскільки всі компоненти пов'язані між собою і будь-які зміни можуть мати широкий вплив.

2. Сервіс-орієнтована архітектура (SOA): У цьому підході система розбивається на сервіси, але вони можуть мати більш монолітну структуру, ніж у мікросервісної архітектури. SOA покладається на використання стандартизованих протоколів для взаємодії між сервісами. В порівнянні з мікросервісною архітектурою, SOA може мати більшу складність у розгортанні та керуванні сервісами, але водночас може бути менш гнучкою та менш незалежною.

3. Інтегрована архітектура: Цей підхід використовує набір компонентів, які працюють разом та спільно виконують функціональні завдання. Він поєднує переваги мікросервісної архітектури та інших архітектурних підходів, проте може мати обмежену гнучкість та масштабованість.

4. Сервер-клієнтська архітектура: Цей підхід використовує розподілену систему, в якій сервери надають послуги, а клієнти взаємодіють з ними. Клієнти зазвичай виконують запити до серверів для отримання даних чи виконання певних операцій. Цей підхід може бути простим у реалізації та керуванні, але він може стати обмеженим у відношенні гнучкості та масштабованості, оскільки всі запити клієнтів обробляються сервером.

5. Компонентна архітектура: У цьому підході система розбивається на окремі компоненти, які виконують конкретні функції. Компоненти можуть бути незалежними та між ними взаємодіяти через спеціальні інтерфейси. Цей підхід дозволяє легко модулізувати та повторно використовувати компоненти, але може потребувати додаткового зусилля для забезпечення взаємодії між компонентами.

6. Контейнеризація та оркестрація: Цей підхід базується на використанні контейнерів, таких як Docker, для упаковки та розгортання окремих компонентів програмного забезпечення. Оркестрація, як наприклад Kubernetes, дозволяє автоматизувати керування та масштабування контейнерами. Цей підхід надає гнучкість та швидкість у розгортанні, масштабуванні та керуванні компонентами, але може потребувати додаткових знань та ресурсів для налагодження інфраструктури.

2.2.2 Вибір мови програмування для розробки додатку

Очевидно, що вибір мови програмування є критичним етапом у розробці, оскільки вона повинна мати відповідні можливості для виконання поставлених задач і бути гнучкою для масштабування сервісу. У моєму конкретному випадку я обрав мову Сі-шарп(позн.C#) та інструменти платформи .NET для розробки додатку.

Мова програмування C# (C-Sharp) має декілька головних особливостей та корисних функцій:

1. Об'єктно-орієнтована природа: C# підтримує концепцію об'єктно-орієнтованого програмування, що дозволяє створювати класи, об'єкти, спадкування, поліморфізм та інші ООП-принципи. Це дозволяє писати структурований та модульний код, полегшує повторне використання та підтримку програм.

2. Керований код та безпека типів: C# генерує "керований код", який виконується в середовищі CLR (Common Language Runtime). Це забезпечує високий рівень безпеки типів та контролю пам'яті, допомагає уникнути помилок виконання та сприяє стабільності програм.

3. Багатопотоковість: C# має вбудовану підтримку багатопотокового програмування. За допомогою класів та бібліотек .NET, можна створювати та керувати паралельними потоками в програмі. Це особливо корисно для розробки програм, які виконують операції вводу-виводу, обчислення або мережеві дії одночасно.

4. Велика стандартна бібліотека: C# має широку стандартну бібліотеку класів, яка надає багато готових рішень для різних завдань. Це включає роботу з файлами, мережами, базами даних, графікою, шифруванням, серіалізацією даних та багато іншого. Використання цих функцій дозволяє розробникам зосередитись на логіці програми, не витрачаючи час на написання базового коду.

5. Кросплатформенність: Однією з важливих переваг мови C# є її кросплатформенність. Завдяки розвитку платформи .NET Core і введенню .NET 5, C# став доступним для розробки на різних операційних системах, таких як Windows, macOS і Linux. Це дозволяє розробникам створювати додатки, які працюють на різних платформах без необхідності повторної розробки або модифікації вихідного коду.

6. Інтеграція з платформою .NET: C# є основною мовою програмування для платформи .NET, що включає в себе велику кількість інструментів, бібліотек та середовищ для розробки додатків. Це забезпечує широкі можливості для розробки різноманітних застосунків, включаючи веб-сайти, веб-служби, настільні додатки, мобільні додатки та ігри.

7. Швидкодія: C# є компільованою мовою, що дозволяє досягати високої швидкодії виконання програм. Завдяки оптимізаціям, які здійснюються в процесі компіляції та виконання коду, програми на C# можуть працювати ефективно та швидко в порівнянні з іншими інтерпретованими мовами програмування.

8. Підтримка розробки інструментів: Мова C# має розширений набір інструментів для розробки, таких як редактори коду (Visual Studio, Visual Studio Code), інтегровані середовища розробки (IDE), відладчики, системи контролю версій та інші. Це полегшує процес розробки, допомагає виявляти й виправляти помилки, покращує продуктивність розробника та сприяє створенню високоякісних програм[22].

Коли мова C# згадується, часто вона асоціюється з технологіями платформи .NET, такими як Windows Forms, WPF, ASP.NET і Xamarin. Навпаки, коли говорять про .NET, часто мають на увазі мову програмування C#. Варто зазначити, що хоча ці терміни пов'язані, вони не є ідентичними. C# була спеціально розроблена для використання в контексті фреймворку .NET, але сама платформа .NET охоплює більш широкий спектр технологій та інструментів[3].

Програми на мові C# виконуються на платформі .NET, яка включає в себе віртуальну систему виконання, називану загальною мовною середовище (Common Language Runtime, CLR), а також набір бібліотек класів. CLR є реалізацією від компанії Microsoft загальної мовної інфраструктури (Common Language Infrastructure, CLI), що є міжнародним стандартом. CLI є основою для створення середовищ виконання та розробки, в яких мови та бібліотеки співіснують безперешкодно.

Вихідний код, написаний на мові C#, компілюється в проміжний мовний код (Intermediate Language, IL), який відповідає специфікації CLI. IL-код та ресурси, такі як бітмапи та рядки, зберігаються у збірці, яка зазвичай має розширення .dll. Збірка містить маніфест, який надає інформацію про типи, версію та культуру збірки.

Під час виконання C#-програми збірка завантажується в CLR. CLR виконує компіляцію Just-In-Time (JIT), перетворюючи IL-код в нативні машинні інструкції. CLR надає також інші сервіси, пов'язані з автоматичним збором сміття, обробкою винятків та управлінням ресурсами. Код, який виконується CLR, іноді називають "керованим кодом". "Некерований код" компілюється в нативну мову машини, яка призначена для певної платформи.

Мовна взаємопов'язаність є важливою особливістю .NET. IL-код, що створюється компілятором C#, відповідає Спільній специфікації типів (Common Type Specification, CTS). IL-код, згенерований з C#, може взаємодіяти з кодом, що був згенерований з .NET-версій F#, Visual Basic, C++. Існує більше 20 інших мов, що також відповідають CTS. Одна збірка може містити кілька модулів, написаних на різних мовах .NET. Типи можуть посилатися один на одного, ніби вони були написані на одній мові.

Крім сервісів виконання, .NET також включає широкий набір бібліотек. Ці бібліотеки підтримують багато різних завдань. Вони організовані у простори імен, які надають різноманітні корисні функції. Бібліотеки включають все, починаючи від роботи з файлами, маніпуляції рядками, розбору XML, веб-фреймворків для веб-додатків та елементів управління

Windows Forms. Типова програма на C# широко використовує бібліотеку класів .NET для роботи з типовими операціями.

Завдяки широкому спектру інструментів та ресурсів, розробка на платформі .NET надає практично необмежені можливості. Хоча існують інші мови програмування, які можуть виконувати подібні завдання, деякі з них спеціалізуються на конкретних областях, а для інших можуть знадобитися сторонні інструменти. У порівнянні з ними, використання C# дозволяє ефективніше та швидше вирішувати широкий спектр завдань з меншими затратами часу та ресурсів. Більш того, мій досвід розробника в роботі з платформою .NET також відіграє важливу роль у виборі цієї мови..

Основні та необхідні можливості та особливості обраної платформи, які були згадані вище, дуже важливі для розробки та подальшого розширення системи. Крім того, якщо вбудованих функцій недостатньо, платформа має зручний менеджер пакетів, що дозволяє легко встановлювати додатковий функціонал.

2.2.3 Патерни проектування

У цьому розділі розглянуті шаблони проектування, які були обрані для використання у системі з різних причин. Деякі з цих шаблонів є необхідною складовою архітектурного рішення і служать засобами його реалізації. Буде висвітлено доцільність використання цих шаблонів, розглянуті сценарії, в яких вони застосовуються, а також переваги, які вони надають[4].

Швидкість розробки та продуктивність програмістів можуть варіюватися залежно від їх рівня вмінь та використовуваних технологій у проектах. У проектуванні програмного забезпечення немає жорстких стандартів або ГОСТів, і розробник сам обирає підхід, яким буде розробляти свою програму. Однак, одним із способів підвищення ефективності роботи є використання шаблону проектування CQRS.

Існують три види патерну CQRS: Regular, Progressive та Deluxe. У нашій роботі ми будемо використовувати класичний патерн Regular CQRS. Progressive і Deluxe є більш складними архітектурами, які вимагають використання більш широкого набору абстракцій.

CQRS (Command Query Responsibility Segregation) - це шаблон проектування (Рисунок 5), який розділяє операції на дві категорії:

- Команди - змінюють стан системи.
- Запити - не змінюють стану, а лише отримують дані.

Цей поділ може бути логічним і базуватися на різних рівнях системи. Крім того, він може бути фізичним і включати різні рівні або компоненти.

Варто зауважити, що CQRS - це не шаблон кодування, а шаблон проектування. Різні компанії використовують цей шаблон по-різному. Його можна використовувати для досягнення кількох цілей:

- Збільшення швидкості розробки нового функціоналу без шкоди для існуючого.
- Зменшення часу, необхідного для приєднання нового розробника до проекту.
- Зменшення кількості помилок.

- Спрощення написання тестів.
- Покращення якості планування розробки.

Завдяки CQRS ми отримуємо архітектуру, в якій все організовано й зрозуміло (менше залежностей, більше сполучень). Розробник може відкрити код команди або запиту, побачити всі його залежності, зрозуміти, що він робить, і продовжити працювати з ним тільки в межах цієї команди або запиту, не занурюючись в інші частини програми[23].

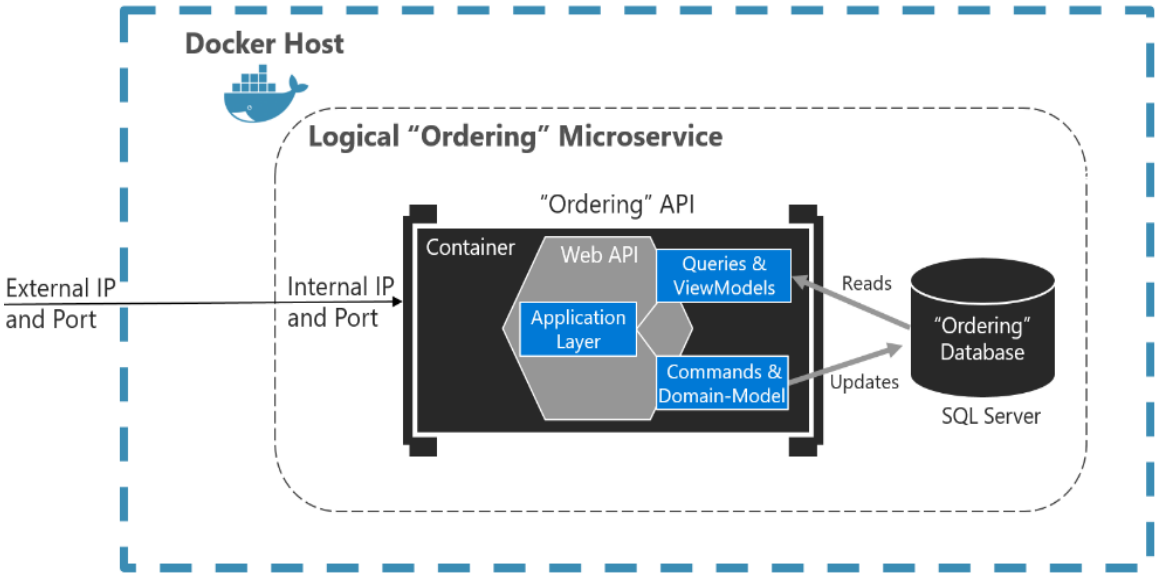

Рисунок 2.5 – приклад архітектури з використанням шаблону [9].

На рисунку 2.5 показано, що на рівні програмного інтерфейсу всередині сервісу відбувається розподіл операцій на запити для читання або запису. У складних системах дані в базі даних можуть бути розподілені на дві категорії: дані, які можна змінювати, і дані, які призначені тільки для зчитування. Ці дані мають готовий вигляд для обробки сервісом.

Шаблон Regular CQRS має свої переваги і недоліки, які варто враховувати при його використанні. Ось деякі з них:

Переваги:

1. Розділення відповідальності: Шаблон дозволяє чітко розділити команди (запити на зміну стану системи) від запитів (запити на отримання даних), що полегшує управління і розуміння системи.

2. Покращена швидкість і масштабованість: Розділення на запити та команди дозволяє оптимізувати і масштабувати систему незалежно для кожного типу операцій. Наприклад, можна використовувати окремі механізми кешування або реплікації для запитів і команд, що поліпшує продуктивність.

3. Зручне тестування: Розділення команд і запитів дозволяє зручно тестувати кожен тип операцій окремо. Це спрощує розробку тестів і забезпечує більшу надійність системи.

4. Гнучкість: Використання CQRS дозволяє змінювати команди і запити незалежно один від одного, що забезпечує гнучкість при розвитку системи і впровадженні нового функціоналу.

Недоліки:

1. Підвищений складність: Використання шаблону CQRS може призвести до підвищеної складності системи, оскільки потрібно керувати розподілом команд і запитів, а також забезпечити синхронізацію між ними.

2. Збільшена затримка: Розділення на команди і запити може призвести до збільшеної затримки в системі, особливо якщо є необхідність у синхронізації даних між різними частинами системи.

3. Підвищений обсяг коду.

4. Підвищена складність розгортання: Використання шаблону CQRS може призвести до підвищеної складності розгортання системи, оскільки потрібно налаштовувати та управляти окремими сервісами для обробки команд і запитів.

5. Навчання і затрати: Впровадження шаблону CQRS може вимагати додаткового навчання для розробників і затрат на розробку нової архітектури та інфраструктури.

Враховуючи ці переваги і недоліки, важливо ретельно обміркувати вибір шаблону CQRS і його застосування в конкретному проекті, враховуючи потреби, обмеження та можливості команди розробників.

Так, можна зрозуміти, що використання шаблону CQRS у спроектованій системі сприяє полегшенню розширення системи та модифікації окремих компонентів. Крім того, цей підхід прискорює робочі бізнес-процеси, оскільки код стає зрозумілим і легко читається навіть для нових розробників.

Застосування шаблону CQRS в системі відбувається шляхом чіткого розподілу відповідальності між компонентами. Це дозволяє застосовувати потрібні технології лише до одного з напрямків функцій системи. Наприклад, можна використовувати кешування запитів або оптимізацію компонентів для покращення продуктивності. Оскільки робота з запитами зосереджена в одній частині програми, а робота з командами в іншій, легко керувати окремими компонентами системи, не зважаючи на деталі, пов'язані з обробкою даних.

Щоб визначити, коли саме потрібно використовувати шаблон CQRS, необхідно повернутися до аналізу предметної області. Це допомагає зробити вибір на користь певної архітектурної парадигми, уникнути зайвої складності системи і забезпечити її гнучкість.

2.2.4 Розробкові фреймворки та бібліотеки

У цьому розділі буде описано набір фреймворків та бібліотек, які були використані під час розробки. Будуть наведені приклади їх використання, розглянуті переваги в порівнянні з іншими технологіями та обгрунтована доцільність їх використання.

Більшість з використаних фреймворків становлять основу для реалізації мікросервісної архітектури, без якої реалізація була б неможливою. Що стосується інших фреймворків, то вони переважно використовувалися для збору статистики про роботу системи або для взаємодії з базою даних.

2.2.4.1 gRPC – протокол

gRPC - це сучасна платформа з відкритим вихідним кодом для високопродуктивного виклику віддалених процедур (RPC), яка працює в будь-якому середовищі. Ця платформа дозволяє ефективно зв'язувати сервіси між собою та з центрами зберігання даних. Вона також надає функції балансування навантаження, відстеження, перевірки справності та аутентифікації.

Ідея gRPC не є новою - цей фреймворк використовує відому парадигму, яка ґрунтується на виклику віддалених процедур (RPC). Застосування RPC включає комунікацію між клієнтом та сервером, для якої використовується не HTTP-виклик, а виклик функції. Клієнт викликає віддалену процедуру, серіалізує параметри та додаткову інформацію у повідомлення і надсилає його на сервер. Після отримання даних сервер проводить їх десеріалізацію, виконує запитану операцію та надсилає результат назад клієнту.

У gRPC клієнтська програма може прямо викликати метод у серверній програмі, розташованій на іншій машині, так, ніби вона працює з локальним об'єктом. Це спрощує розробку розподілених програм та служб (Рисунок 2.6). Схоже на багато інших систем RPC, gRPC ґрунтується на ідеї визначення служби, де вказуються методи, які можна віддалено викликати з їх

параметрами та типами повернення. На стороні сервера, сервер реалізує цей інтерфейс та запускає сервер gRPC для обробки викликів від клієнта. На стороні клієнта, клієнт має заглушку, яка надає ті самі методи, що і сервер[7].

Використання передових технологій, які мають високу продуктивність порівняно з JSON і XML, а також забезпечують більшу цілісність API, є причиною створення та популярності gRPC.

Протоколи буферів, також відомі як Protobuf, є стандартами серіалізації або десеріалізації, які спрощують визначення додатків та автоматично генерують код клієнтських бібліотек. Остання версія, proto3, є більш простою у використанні та пропонує нові можливості gRPC.

Файли .proto забезпечують роботу служб gRPC та обмін між клієнтами gRPC та повідомленнями сервера. Файл .proto завантажується в пам'ять під час виконання компілятором Protobuf - protoc. Цей компілятор створює графічні клієнтські та графічні серверні додатки gRPC, які використовують структуру в пам'яті для серіалізації та десеріалізації бінарних даних. Кожне повідомлення надсилається та приймається між користувачем та віддаленою службою після генерації коду в gRPC[24].

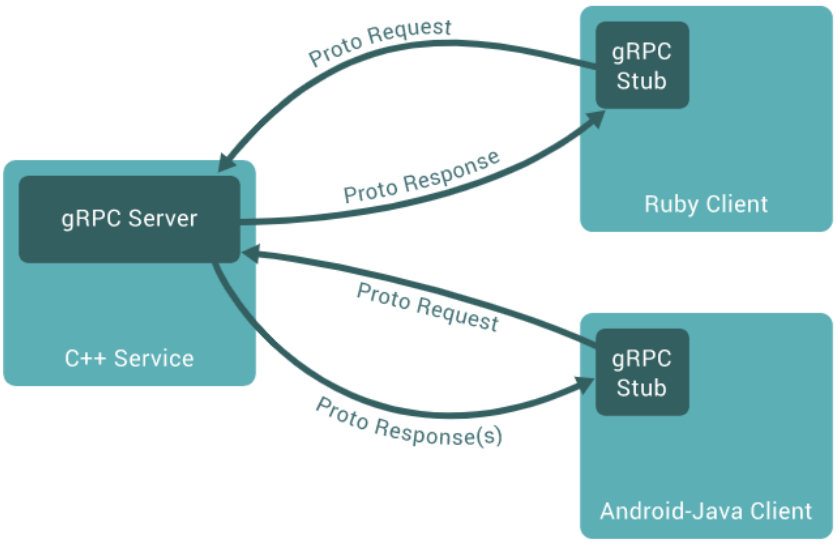

Рисунок 2.6 – ілюстрація роботи сервісів створених на різних мовах програмування[10].

Балансування навантаження відбувається на стороні клієнта (Рисунок 2.7). Клієнт використовує простий алгоритм "round-robin", щоб розподілити запити між серверами зі списку, отриманого від сервера. За бажанням, на стороні сервера можна налаштувати складніші алгоритми для формування списку back-end сервісів, використовуючи політики балансування навантаження.

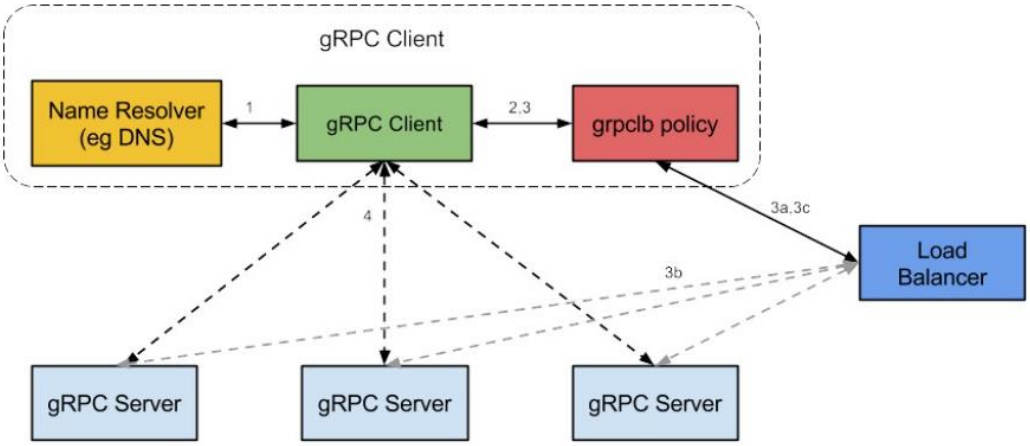

Рисунок 2.7 – схема балансування навантаження [11].

Отже, gRPC переважно застосовується для розробки внутрішніх систем, що означає створення інфраструктури, котра не доступна зовнішнім користувачам.

2.2.4.2 Entity Framework Core

Ядро фреймворку сутностей(*Entity Framework Core)* – є легким, розширюваним та об'єктно-орієнтованим технологічним рішенням від компанії Microsoft для доступу до даних. EF Core є ORM-інструментом (відображення об'єктів на реляційні дані). Іншими словами, EF Core дозволяє працювати з базами даних на вищому рівні абстракції: він дозволяє абстрагуватись від самої бази даних та її таблиць, щоб працювати з даними незалежно від типу сховища. На фізичному рівні ми маємо справу з таблицями, індексами, первинними та зовнішніми ключами, але на

концептуальному рівні, який надає нам Entity Framework, ми вже працюємо з об'єктами[25].

Entity Framework Core підтримує багато різних систем управління базами даних (СУБД). Таким чином, ми можемо використовувати EF Core з будь-якою СУБД, якщо є відповідний постачальник.

На даний момент, за замовчуванням, Microsoft надає деякі вбудовані постачальники: для роботи з MS SQL Server, SQLite та PostgreSQL. Також існують постачальники від сторонніх вендорів, наприклад, для MySQL.

Також варто відзначити, що EF Core надає універсальний API для роботи з даними. Якщо, наприклад, ми вирішимо змінити цільову СУБД, основні зміни в проекті будуть стосуватись переважно конфігурації та налаштування підключення до відповідних провайдерів. Але код, який безпосередньо працює з даними, отримує їх, додає до бази даних і т.д., залишиться незмінним.

Entity Framework Core (EF Core) має кілька переваг, серед яких:

1. Легкість використання: EF Core надає простий API для роботи з базами даних, що полегшує розробку і підтримку додатків.

2. Абстракція від бази даних: EF Core дозволяє абстрагуватись від конкретної СУБД і працювати з даними на вищому рівні об'єктів. Це дозволяє змінювати СУБД без необхідності внесення значних змін у коді додатка.

3. Підтримка різних СУБД: EF Core підтримує багато різних систем управління базами даних, що дозволяє використовувати його з різними типами сховищ даних.

4. Міграції бази даних: EF Core надає механізм міграцій, який спрощує розгортання та оновлення схеми бази даних без втрати даних.

5. Підтримка LINQ: EF Core інтегрується з мовою C# і надає підтримку Language-Integrated Query (LINQ). Це дозволяє зручно і ефективно виконувати операції з даними за допомогою звичайних мовних конструкцій.

6. Підтримка асинхронної розробки: EF Core надає підтримку асинхронних запитів до бази даних, що поліпшує продуктивність додатка та масштабованість.

7. Розширюваність: EF Core можна розширювати за допомогою власних провайдерів та власних функцій маппінгу.

Ці переваги роблять EF Core потужним і зручним інструментом для розробки додатків, що взаємодіють з базами даних.

Цей фреймворк був обраний з метою скористатись перевагами, згаданими вище. Використання EF Core з підходом "Code First" дозволяє економити час у початковій фазі проекту, що сприяє швидкішому розвитку системи[6].

2.2.4.3 Serilog + .Net

Однією з зручних бібліотек .NET, які підтримують структуроване ведення журналу, є Serilog. Ця бібліотека підтримує всі основні функції ведення журналу, які присутні у log4net, Nlog та інших відомих бібліотеках.

Кілька загальноприйнятих типів запису:

Verbose (Докладний) - найнижчий рівень детального ведення журналу (наприклад, передані аргументи у метод).

Debug (Налагодження) - дані для налагодження коду, вищий рівень за Verbose (наприклад, який метод викликали і його результат).

Warning (Попередження) - попередження для бізнес-процесу, не повинно містити Debug-дані (наприклад, запустили розрахунок зарплати).

Error (Помилка) - помилка в додатку, яку не очікували.

Fatal (Критична помилка) - виняткова помилка, яка зупиняє бізнес-процеси додатку (наприклад, перенаправлення користувача до PayPal, а сума платежу покупця не відповідає очікуваній).

Різні типи накопичувачів (стоків) даних в Serilog: текстовий файл, реляційні бази даних, NoSql бази даних, події Windows, http/s запити і т.д.

Зручна конфігурація як через код, так і через .config файли.

Уся інформація про роботу системи може бути виведена як на консоль розробника, так і збережена у окремому файлі. Порівняно з іншими інструментами ведення журналу, Serilog пропонує більше можливостей для форматування тексту і має покращену структуру подій, що полегшує їх зрозуміння.

Опишу ще про кілька переваг Serilog, які роблять його привабливим в якості інструменту логування:

1. Гнучкість форматування: Serilog надає більше варіантів для форматування лог-повідомлень, дозволяючи налаштувати вигляд і структуру журнальних записів згідно з власними потребами.

2. Багатоцільовість: Serilog підтримує різноманітні вихідні носії (стоки), такі як текстові файли, бази даних реляційного та NoSQL типу, події Windows, HTTP-запити тощо. Це дає можливість зберігати лог-дані у різних форматах та середовищах відповідно до вимог проекту.

3. Простота конфігурації: Serilog надає зручні засоби для налаштування через код або конфігураційні файли. Це дозволяє легко змінювати параметри логування без необхідності внесення змін у самому коді додатку.

4. Структурованість подій: Serilog дозволяє організовувати журнальні записи у структуровані події з ключ-значеннями. Це полегшує аналіз та обробку лог-даних, їх фільтрацію та пошук за певними критеріями.

5. Розширюваність: Serilog можна розширити за допомогою різних плагінів та додаткових модулів, що дозволяє розширити його функціональність або інтегрувати з іншими інструментами та системами.

Загалом, Serilog є потужним та гнучким інструментом логування, який надає широкі можливості для контролю та аналізу діагностичної інформації у додатках.

2.2.5 Вибір СУБД

В якості системи управління базами даних було вибрано PostgreSQL.

PostgreSQL - потужна реляційна база даних відкритого джерела рівня підприємства. Вона дозволяє використовувати як реляційні SQL, так і нереляційні JSON дані та запити. PostgreSQL має сильну спільноту. Це надійна система управління базами даних з високим рівнем підтримки, безпеки і точності. Деякі мобільні та веб-додатки використовують PostgreSQL як базу даних за замовчуванням. Багато геопросторових і аналітичних рішень також використовують PostgreSQL. Остання версія - PostgreSQL 15[26].

PostgreSQL підтримує складні типи даних. Фактично, ця база даних була створена з урахуванням великої кількості типів даних.

Однією з причин популярності PostgreSQL є його набір функцій. Ця база даних допомагає в розробці додатків, забезпечуючи цілісність даних. Вона дозволяє адміністраторам створювати стійкі до відмов середовища. Крім того, вона може використовуватися на різних платформах і підтримує всі популярні мови програмування.

База даних також надає дуже розвинуту систему блокування. Вона також має контроль паралелізму в декількох варіантах. Сервер бази даних PostgreSQL також має можливості для зрілого програмування на серверній стороні. Він відповідає стандарту ANSI SQL і повністю підтримує мережеву архітектуру клієнт-сервер.

PostgreSQL також має високу доступність і резервний сервер. Він відповідає стандарту ANSI-SQL2008 і є об'єктно-орієнтованим. Можливість з'єднання з іншими сховищами даних, такими як NoSQL, яке служить єдиним центром для поліглотних систем, забезпечується завдяки підтримці базою даних JSON. Інформацію саме одного кластера баз даних завжди керує один

екземпляр PostgreSQL. Кластер баз даних - це група записів, які зберігаються в одному місці файлової системи.

Також PostgreSQL може похвалитись такими перевагами:

1. Надійність і стабільність: PostgreSQL є високопродуктивною та надійною базою даних з високим рівнем цілісності і стійкості до відмов.

2. Гнучкість та розширюваність: PostgreSQL має великий набір вбудованих типів даних і можливостей для визначення користувацьких типів і функцій. Вона також підтримує розширення, які дозволяють додавати нові функції і можливості до бази даних.

3. Підтримка SQL та ACID: PostgreSQL повністю відповідає стандарту SQL і забезпечує ACID (атомарність, консистентність, ізольованість, довереність) властивості для забезпечення цілісності та безпеки даних.

4. Підтримка реплікації та резервного копіювання: PostgreSQL має вбудовані засоби для створення резервних копій, реплікації та відновлення даних, що дозволяє забезпечити високу доступність та захист від втрати даних.

PostgreSQL пропонує безліч можливостей. Як база даних, побудована на об'єктно-реляційній моделі, вона підтримує складні структури та широкий спектр вбудованих та користувацьких типів даних. PostgreSQL володіє великим обсягом даних та має репутацію надійності і забезпечення цілісності даних. Хоча можливо, не всі розширені можливості зберігання даних будуть необхідними у всіх випадках, наявність всіх цих функцій забезпечує перевагу, оскільки потреби можуть зростати швидко і бути передбачені заздалегідь.

3 СТВОРЕННЯ СИСТЕМИ

В цьому розділі розглядаються процеси формування програмного коду і встановлення структури системи, а також опис використання додаткових бібліотек та фреймворків.

3.1 Компоненти додатку

Після визначення архітектурного рішення, мови розробки та інструментів реалізації, а також після розробки всіх необхідних сценаріїв, можна приступити до структуризації системи та її окремих компонентів. На рисунку 3.8 наведена структурна схема розроблюваного сервісу, яка відображає результати аналізу предметної області та виділені сутності, що будуть використовуватись в роботі сервісу.

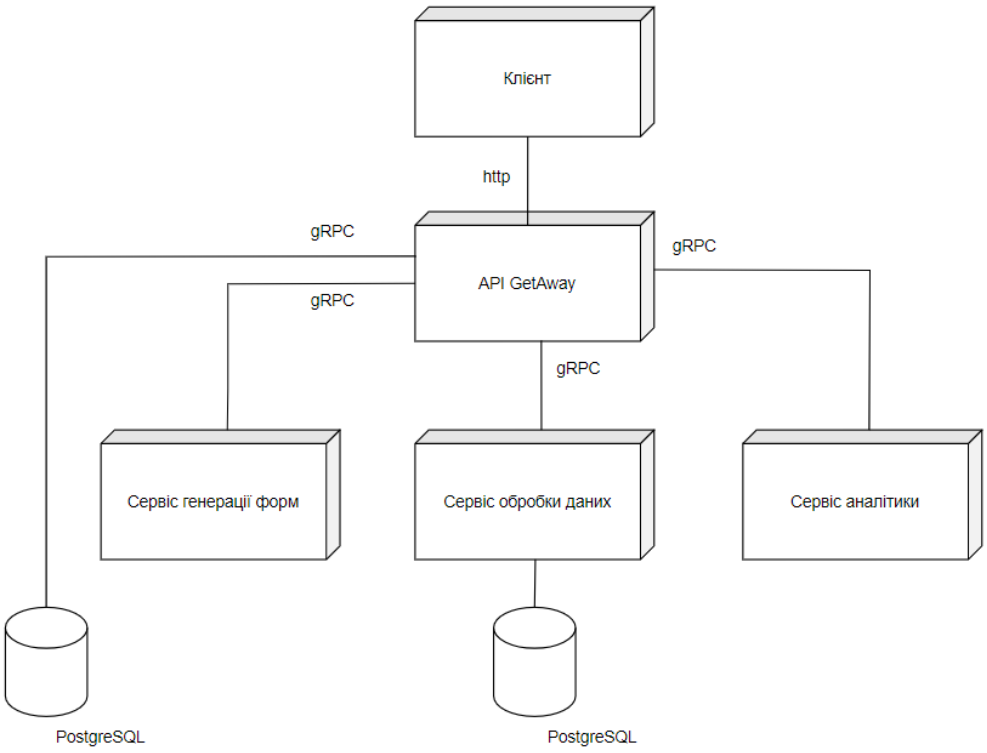

Рисунок 3.8 – Структура системи

Структурна схема сервісу, що описується в даній роботі, включає наступні компоненти:

1. API Gateway: Відповідає за маршрутизацію запитів та виконує функції реєстрації та авторизації користувачів.

2. Сервіс обробки даних: Займається обробкою даних, що надходять з зовнішніх джерел, та зберіганням їх у базі даних.

3. Сервіс генерації форм: Включає механізм генерації форм на основі вхідних даних.

4. Сервіс аналітики: Відповідає за створення аналітичних звітів на основі даних, отриманих з зовнішніх джерел.

5. Клієнт: Забезпечує інтерфейс для взаємодії користувача з додатком, включаючи авторизацію, створення форм та перегляд аналітики.

У мікросервісній архітектурі кожен сервіс має своє власне сховище даних, до якого має доступ тільки він сам. Якщо потрібно оновити дані в декількох сервісах одночасно, дані передаються до маршрутизатора, який пересилає їх до потрібного сервісу.

Особливістю такого підходу до архітектури є те, що вона не має незамінюваних компонентів. Якщо потрібно збільшити навантаження на певний сервіс, можна просто розгорнути додаткові копії цього сервісу на сервері.

Для обміну інформацією використовуються протокол HTTP і стандарт gRPC. gRPC визначає чіткі межі між компонентами програми і надає опис методів їх взаємодії[5].

В компонентах системи застосовано принципи чистої архітектури SOLID, які передбачають розмежування відповідальностей та розташування їх у відокремлених частинах з ієрархічною структурою. В цій структурі сутності, які не залежать одна від одної, встановлюють правила, що визначають поведінку системи.

Принцип чистої архітектури встановлює чіткі вимоги до структури компонентів, яка має такі складові:

1. Сутності: Вони містять поля, які описують об'єкти предметної області.

2. Варіанти використання: Ці компоненти реалізують логіку роботи системи і використовують сутності для цього.

3. Репозиторії: Вони виступають посередниками між сховищем даних і обробником, забезпечуючи доступ до даних.

4. Інфраструктура: Цей компонент містить механізми, які організовують доступ до бази даних.

5. Інтерфейс: Це інтерфейс користувача, через який відбувається взаємодія клієнта з серверною частиною.

В основі такого підходу лежать принципи концепції MVC, яка дозволяє розмежовувати відображення, логіку та збереження даних в системі. Однак, в обраній архітектурі існує відмінність, оскільки базується вона на механізмах бази даних, що робить її заміну важкодоступною. Основний акцент чистої архітектури зосереджений на потребах предметної області та сутностях. Однією з переваг є відсутність залежності від компонента, який може бути замінений при потребі.

Така структура сприяє ефективній командній розробці, оскільки дозволяє призначати різні команди для роботи над різними частинами системи. Структура системи має ієрархічний характер, що означає, що реалізація функціоналу не є обов'язковою, а достатньо мати інформацію про інтерфейс взаємодії між сервісами, що надається відповідною командою розробників.

Отже, можна зробити висновок, що команда розробників не повинна мати детального розуміння всієї системи, а має фокусуватись на функціоналі, що знаходиться у їхній відповідальності.

3.2 Тестування

Важливо включати тестування під час розробки, оскільки написаний код повинен бути належним чином протестованим. Це дозволяє перевірити, чи працює функціонал належним чином і не впливає на вже наявний функціонал, який був розроблений раніше. Тестування в процесі розробки допомагає виявити та усунути помилки та проблеми з роботою коду, а також забезпечує рівномірне покриття функціоналу тестами.

Для тестування сервісу використовуються два типи тестів:

1. Тести сценаріїв, які описують різні сценарії роботи системи з вказанням вхідних та вихідних даних, а також спеціальних умов, таких як некоректні дані. Ці тести допомагають виявити проблеми та перевірити коректність роботи системи у різних ситуаціях. За допомогою цих сценаріїв можуть бути створені модульні тести.

2. Модульні тести, які перевіряють правильність роботи окремих методів програми. Ці тести включають в себе покриття всіх методів системи та перевіряють їх коректну роботу.

Такий вибір інструментів для тестування обумовлений складністю розроблюваної системи, яка складається з багатьох мікросервісів, кожен з яких має складну логіку.

Модульні тести використовуються для перевірки правильності роботи методів з різними вхідними даними.

Загалом, такі тести мають схожу структуру, яка включає наступні етапи:

1. Підготовка даних: визначення різних варіантів даних, які будуть використовуватися під час тестування.

2. Виклик методу: виклик методу з використанням підготовлених даних.

3. Перевірка результату: порівняння очікуваних даних з отриманим результатом роботи методу.

Таким чином, структура тестів передбачає підготовку даних, виклик методу та перевірку правильності результату.

Під час внесення змін або додавання нового функціоналу до програми, важливим кроком є запуск тестів для перевірки правильності роботи програми. Цей процес забезпечує розробника впевненість у коректності функціоналу. У разі виявлення помилок, непройдений тест точно вказує на метод, в якому виникла проблема. Це спрощує аналіз проблеми і її виправлення

3.3 Веб-інтерфейс сервісу

Для розробки веб-інтерфейсу користувача був використаний фреймворк React та мова програмування JavaScript. Для створення вигляду сторінок та їх стилізації використовувалися HTML і CSS.

Головна мета використання фреймворку React полягала у розробці односторінкових додатків (SPA). Такі додатки складаються з однієї динамічної сторінки, що дозволяє перемикатися між різними розділами без перезавантаження сторінки.

Вибір фреймворку був обґрунтований поширеністю мови програмування JavaScript. Практично всі сучасні веб-браузери мають вбудований компілятор JavaScript або підтримують роботу з цією мовою. Інтеграція з JavaScript є ключовим фактором вибору, оскільки дозволяє вписувати JavaScript-код безпосередньо в HTML, що спрощує процес розробки динамічних сторінок і не вимагає використання додаткових інструментів.

Серед переваг фреймворку React можна виділити наступні особливості:

1. Віртуальна об'єктна модель документа забезпечує ефективність та високу продуктивність. Це означає, що зміни відображення елементів відбуваються швидко та оптимально.

2. Повторне використання компонентів є однією з ключових переваг React. Розробники можуть створювати багаторазові компоненти, які легко перевикористовувати в різних частинах додатку. Крім того, існує велика бібліотека готових компонентів, які можна використовувати безпосередньо.

3. Потік даних "зверху - вниз" (від батьківських компонентів до дочірніх) спрощує передачу даних між компонентами і запобігає непередбаченим помилкам. Цей однобічний потік даних полегшує налагодження та розуміння того, як дані передаються та змінюються в додатку.

4. Фреймворк React надає браузерні інструменти для розробників, які допомагають у тестуванні додатків. Ці інструменти включають набір віджетів для тестування, які допомагають виявляти та виправляти проблеми в додатку.

4 ОГЛЯД ПРАКТИЧНОГО ЗАСТОСУВАННЯ ВЕБ-СЕРВІСУ

4.1 Чим є веб-сервіс

Розроблений для практичного застосування веб-сервіс, являє собою landing-сторінку. Як описувалось вже у попередніх розділах - лендінг, також відомий як цільова сторінка, є компактним веб-сайтом, спеціально розробленим для досягнення конкретної цілі. Зазвичай, лендінг складається з однієї сторінки, на якій розміщена вся необхідна інформація про товар або послугу, яка має привернути увагу відвідувачів та спонукати їх до виконання певної дії.

Головна мета лендінгу полягає в тому, щоб зацікавити відвідувача та перетворити його на потенційного клієнта або здійснити іншу цільову дію. В залежності від цілей, лендінг може пропонувати товари або послуги, заохочувати передплату на розсилку або виконувати іншу певнудію.

Ефективність лендінгу зумовлена психологією: він повинен бути привабливим та захоплюючим, щоб ефективно привернути увагу відвідувача. Тому для створення успішного лендінгу необхідно враховувати певні вимоги. Якщо сторінка не буде добре продуманою, буде складніше збільшити конверсію.

Робочий лендінг має цікавий дизайн, переконливий текст, стратегічно розміщені контакти компанії і заклик до дії. Для розробки успішного лендінгу важливо розуміти принципи людської психології та використовувати прийоми, які переконують відвідувача залишитися на сторінці. Лендінг прагне підштовхнути потенційного клієнта до дії та перетворити його на активного клієнта.

4.2 Підстави для вибору лендінгу

Landing-сторінка не є універсальним рішенням для всіх видів веб-сервісу. Її використання залежить від конкретних потреб і завдань. В деяких випадках лендінг буде достатнім, а в інших випадках краще використовувати звичайний багатосторінковий сайт.

Landing-сторінка найбільш підходить для ситуацій, коли мається на увазі:

- Продаж недорогих товарів у невеликій кількості. Оскільки лендінгова сторінка має обмежену кількість інформації, вона непрактична для розміщення широкого асортименту товарів. В таких випадках краще використовувати інтернет-магазин.

- Пропозиція одного товару або послуги з можливістю зміни характеристик. Наприклад, це можуть бути онлайн-курси англійської мови або асортимент органічних солодощів без цукру.

- Пропозиції, які вже є популярними та відомими клієнтам з інших джерел. В таких випадках не потрібно детально описувати товар чи послугу, а достатньо правильно сформульованої пропозиції, щоб привернути увагу і залучити клієнтів.

- Збір бази клієнтів. Якщо вам потрібно формувати базу даних клієнтів і залучати нових користувачів, лендінг може бути ефективним інструментом для збору контактних даних зацікавлених відвідувачів.

Landing-сторінка не є найкращим варіантом в таких ситуаціях:

1. Якщо ваш продукт або послуга вже відомі клієнтам, а в галузі є багато конкурентів. В цьому випадку краще створити повноцінний веб-сайт, де можна докладно описати вашу пропозицію та переваги вашої компанії.

2. Якщо ваша мета полягає у розповіді про компанію та немає потреби отримувати від відвідувачів конкретні дії. В такому випадку краще використовувати блог або об'ємний інформаційний сайт, де можна більш детально розкрити інформацію про компанію та її діяльність.

3. Якщо ваша мета полягає у створенні та розвитку бренду компанії. Лендінг пейдж зазвичай має меншу довіру у користувачів порівняно з багатосторінковим сайтом. Якщо ваша стратегія передбачає довгостроковий розвиток бренду, то рекомендується не починати з лендінгу, а створити комплексний веб-сайт, що буде докладніше відображати вашу компанію та її цінності.

В кожному з цих випадків варто уважно аналізувати цілі та потреби вашого бізнесу, а також розуміти свою цільову аудиторію, щоб визначити найкращий підхід до створення веб-сервісу.

Тому уважно проаналізувавши усі наведені вище пункти, було прийнято рішення про використання Landing-сторінки, а не комплексного веб-сайту.

4.3 Інтерфейс веб-сервісу та демонстрація роботи

Перш за все слід зауважити, що розроблений застосунок направлений на користувачів з Європейських та Американських країн.

При потраплянні на веб-сервіс, ми одразу бачимо перед собою такий екран, котрий містить 2 основні кнопки (Рисунок 4.9).

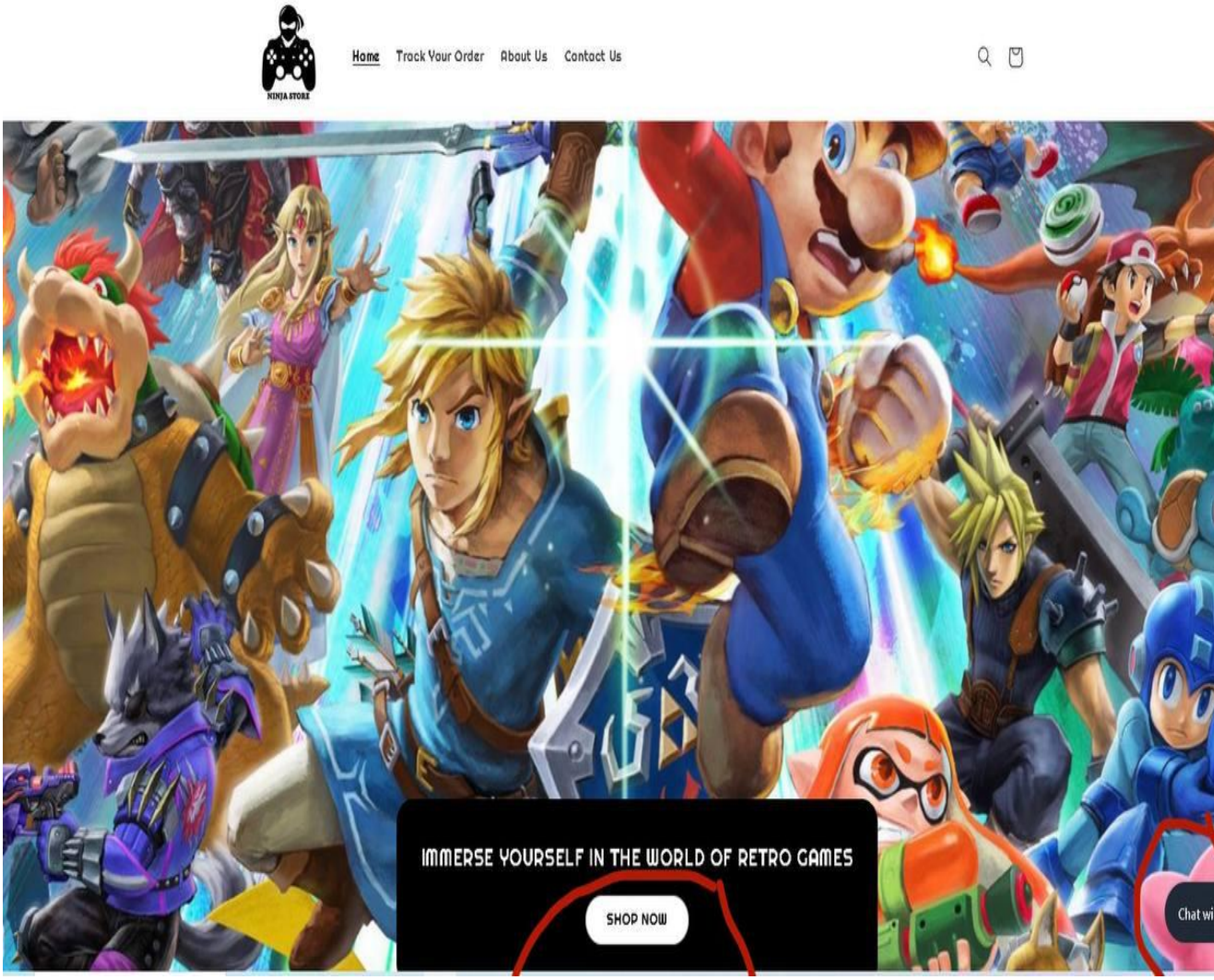

Рисунок 4.9- Інтерфейс сайту

Перша кнопка “Chat with us” – надає змогу користувачу задати будь-яке запитання, без уточнення своєї персональної інформації. Друга кнопка “Shop now” – перемістить користувача до наступної частини веб-сервісу, де він зможе побачити повний перелік продуктів, що надаються (Рисунок 4.10).

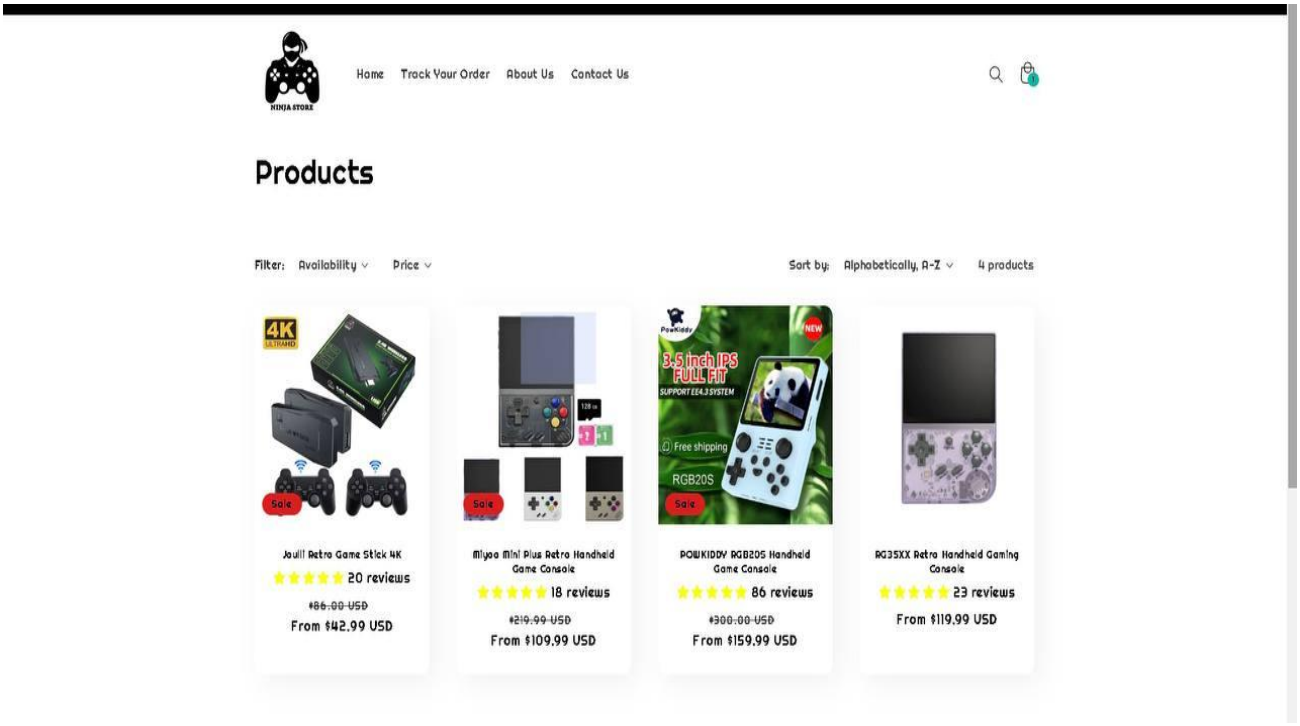

Рисунок 4.10

Якщо користувач зацікавився продуктом, то він має змогу отримати повний огляд продукту та перейти до оформлення замовлення (Рисунок 4.11).

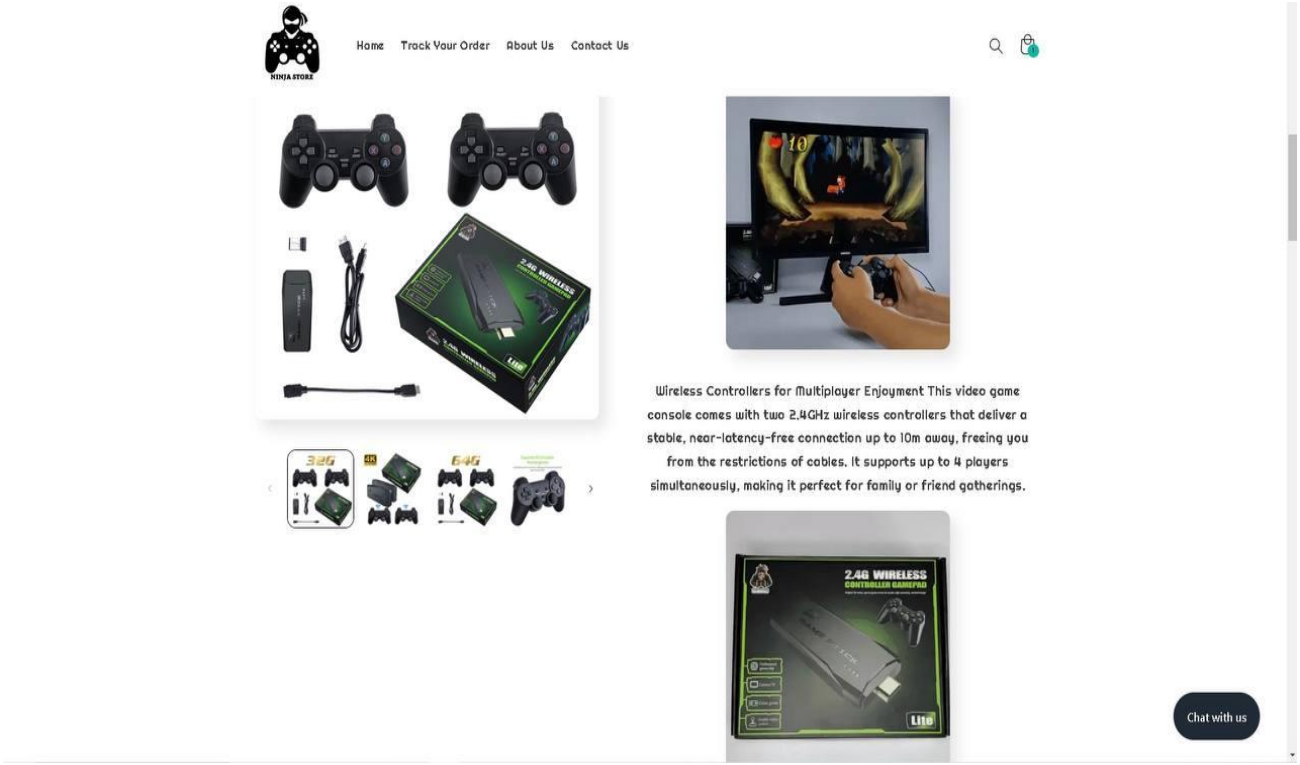

Рисунок 4.11

При оформленні замовлення користувача зустріне веб-форма, в котру він має ввести свої персональні дані, аби мати можливість для зворотного

зв’язку(FeedBack)(Рисунок4.12).

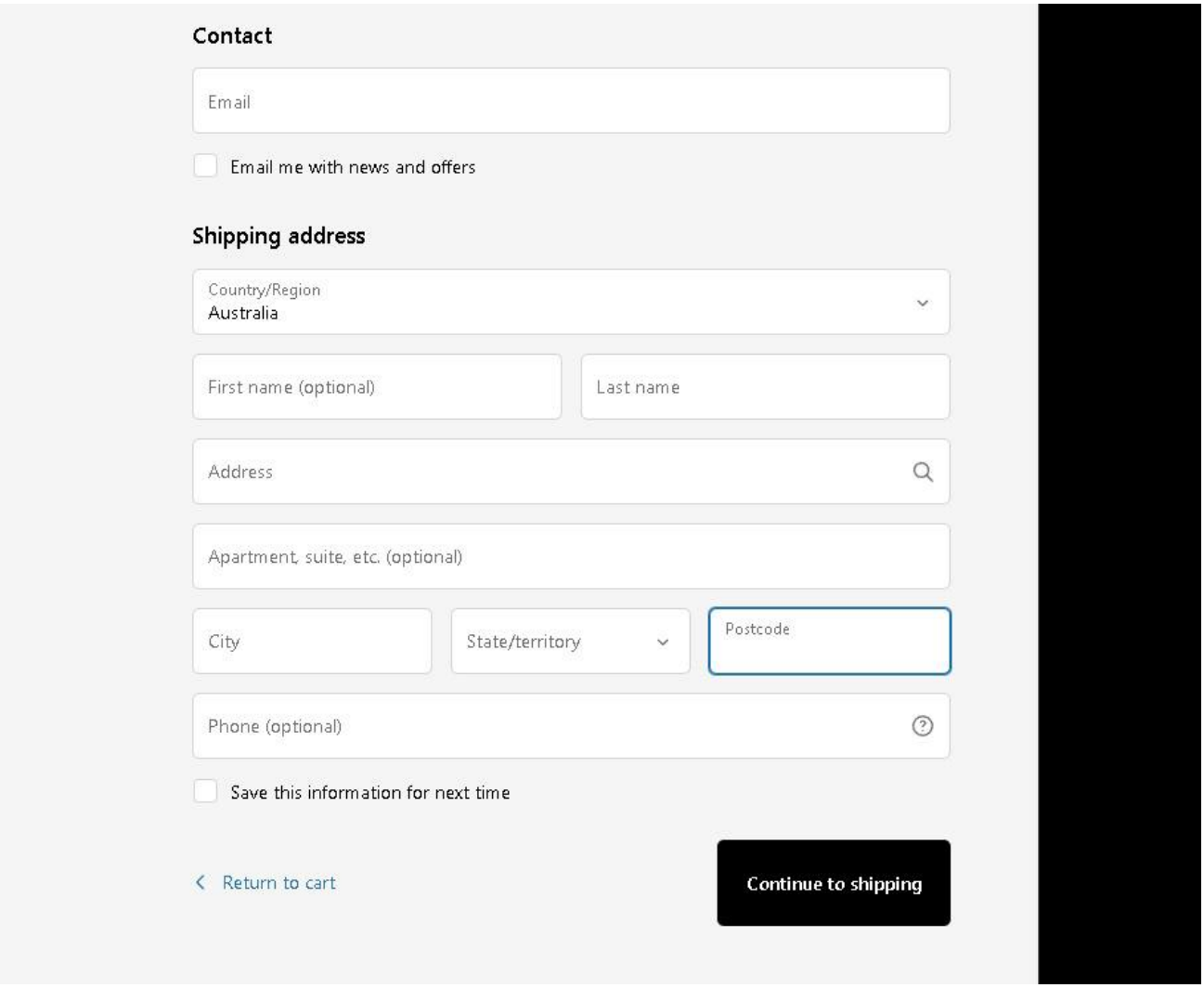

Рисунок 4.12

Важливим є те, що інформація, яку надає користувач, по –перше буде конфіденційною(тобто доступ до неї зможуть мати лише користувач та власник форми) , а по-друге буде перевірятись на її актуальність під час стадії записування (наприклад , при виборі країни Іспанія, користувач не зможе помилитись і вказати поштовий індекс якоїсь іншої країни). Після заповнення форми, вся подана користувачем інформація буде занесена до бази даних (Рисунок 4.13).

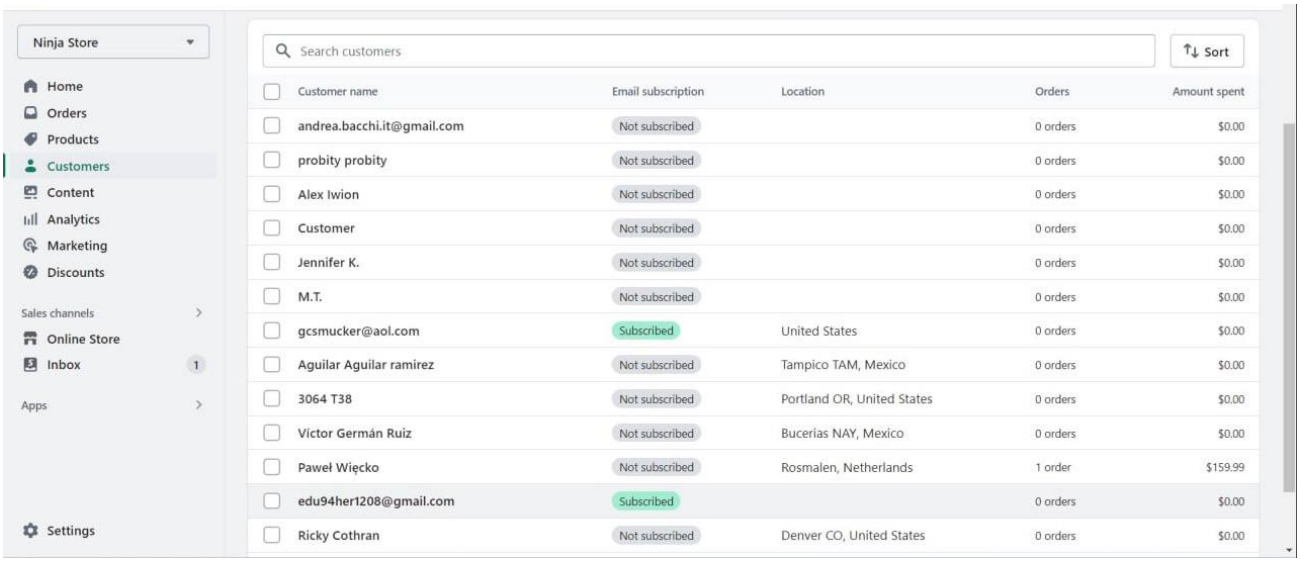

Рисунок 4.13

Далі вже власник веб-сервісу може зв’язатись з клієнтом задля уточнення деяких деталей замовлення.

Слід також наголосити, що також є можливість проводити безліч операцій з отриманими даними. Наприклад, провести аналіз кількості переглядів сайту, оцінки користувачів по якості діалогу з оператором, кількості та суми придбань що були здійсненні, тощо(Рисунок 4.14).

Вся ця інформація буде подана у вигляді графіків та діаграм.

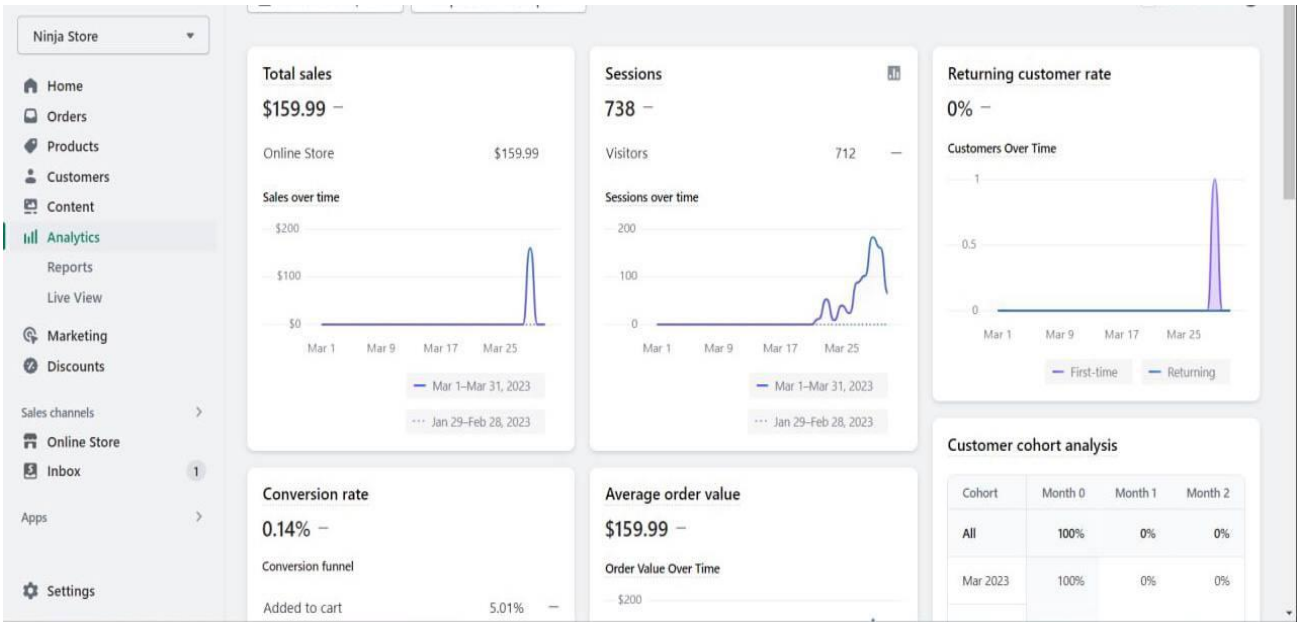

Рисунок 4.14

Отже можна зробити висновок, що додаток є повністю функціонуючим, база даних якого заповнена актуальною інформацією від користувачів , і за потреби має здатність до удосконалення і оновлення.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було успішно розроблено та отримано повністю функціонуючий сервіс для збереження та управління даними. Цей сервіс включає функцію генерації форм та формування аналітики на основі наданих даних. Він повністю задовольняє вимоги та відповідає виявленим потребам, що були визначені під час аналізу існуючих рішень у даній предметній області.

Перед розпочатком основної фази розробки проекту було налагоджено робоче середовище, встановлено всі необхідні пакети та ресурси, створено репозиторій у системі контролю версій і перевірено відповідність до системних вимог.

На початковому етапі конструювання проекту були визначені сутності, що відносяться до предметної області, на основі яких були створені сценарії роботи системи. Також були оцінені переваги обраного архітектурного стилю, що визначив спрямування розвитку проекту. Були обрані засоби розробки, які забезпечують необхідні інструменти для реалізації системи. Крім того, були вибрані додаткові бібліотеки, які сприятимуть прискоренню розробки та спрощенню коду. Нарешті, була обрана СУБД, яка найкращим чином відповідає описаним вимогам розроблюваної системи.

Маючи ці дані, можна приступати до розробки системи. Після завершення цього етапу стало зрозумілим, що він є критичним усій процедурі розробки, оскільки на його основі можна спланувати подальший розвиток системи. Планування також дозволяє передбачити та запобігти можливим помилкам, що можуть виникнути у майбутньому.

Наступним кроком було структурування системи в цілому і кожного її окремого компонента. Для цього був використаний підхід чистої архітектури, який передбачає розбиття додатку на компоненти з чіткою ієрархічною структурою, що базується на сутностях предметної області та правилах, яким

вони підпорядковуються. Це дозволило досягти ізольованості компонентів, можливості їх повторного використання та поліпшення читабельності коду.

Для реалізації веб-інтерфейсу було обрано відповідний фреймворк, оскільки мова програмування, на якій він базується, є стандартною для більшості браузерів. Це дозволило створювати динамічні та зручні сторінки, використовуючи вбудований функціонал.

Використовуючи описані технології, були виконані вимоги до масштабованості та гнучкості системи, забезпечено її простоту та зрозумілість.

Також було практично оглянуто розроблений веб-сервіс, як готовий існуючий сайт.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Microservices. URL: https://habr.com/ru/articles/249183/

2. .NET Microservices: Cesar de la Torre, Bill Wagner, Mike Rousos, 2020

3. Pro C# 7: With .NET and .NET Core, Andrew Troelsen, 2019

4. Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994

5. Clean architecture: A Craftsman`s Guide to Software Structure and Design, Robert Martin, 2017

6. Entity vs Value Object: the ultimate list of differences, Vladimir Khorikov, 2016

7. gRPC specification. URL: https://grpc.io/docs/

8. Introduction to microservices. URL: https://www.nitendratech.com/programming/introduction-to-microservices/

9. Apply CQRS patterns. URL: https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/applysimplified-microservice-cqrs-ddd-patterns

10. gRPC client-server communication. URL: https://ru.m.wikipedia.org/wiki/Файл:Grpc-client-server-communication.svg

11. Inter-Service communication with gRPC. URL: https://medium.com/inter-service-communication-with-grpc-d815a561e3a1

12. Всі можливості Google Forms. URL: https://web-promo.ua/blog/kratkij-gajd-vse-vozmozhnosti-google-forms/

13. Що таке Landing Page і для чого він потрібен. URL: https://softmaks.com/landingpage.html

14. Інформація про FormDesigner. URL: https://www.tadviser.ru/index.php/%D0%9F%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82:FormDesigner#:~:text=FormDesigner%20%2D%20%D0%BE%D0%BD%D0%BB%D0%B0%D0%B9%D0%BD%2D%D0%BA%D0%BE

%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80%20%D0%B2%D0%B5%D0%B1%2D,%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80%D0%B0%20%D0%B2%D1%8B%D1%81%D1%82%D1%83%D0%BF%D0%B0%D0%B5%D1%82%20%D0%BF%D1%80%D0%B5%D0%B4%D0%BF%D1%80%D0%B8%D0%BD%D0%B8%D0%BC%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%20%D0%98%D0%B2%D0%B0%D0%BD%20%D0%A8%D0%B0%D0%BC%D1%88%D1%83%D1%80.

15. About QuintaDB. URL: https://sourceforge.net/software/product/QuintaDB/

16. System Modeling and Analysis: a Practical Approach Gerrit Muller University of South-Eastern Norway-NISE Hasbergsvei 36 P.O. Box 235, NO-3603 Kongsberg Norway, 2021.

17. Київський професійно-педагогічний коледж імені Антона Макаренка – “Електронний навчально-методичний посібник ” Інформатика та Комп. Техніка. URL: https://kppk.com.ua/ELLIB/ebook/Gorbenko/IKT/10/10.htm

18. Що таке Use Case та для чого вони потрібні,2021. URL: https://training.qatestlab.com/blog/technical-articles/what-is-a-use-case-and-what-are-they-for/

19. Мікросервіси та мікросервісна архітектура. URL: https://www.atlassian.com/ru/microservices

20. Типи мережевих протоколів та їх призначення. URL: https://deltahost.ua/ua/tipi-merezhevix-protokoliv-i-ih-priznachennya-http-ip-ssh-ftp-pop3-mac.html

21. Переваги та недоліки використання мікросервісної архітектури при розробці програмного забеспечення, 2021. URL: https://conf.ztu.edu.ua/wp-content/uploads/2019/12/31-1.pdf

22. A tour of the C# language, 2023. URL: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

23. Патерн CQRS, 2021. URL: https://habr.com/ru/articles/543828/

24. ІЩо таке gRPC?, 2022. URL: https://appmaster.io/ru/blog/chto-takoe-gpk

25. ІЩо таке Entity Framework Core?, 2021. URL: https://metanit.com/sharp/entityframeworkcore/1.1.php

26. ІЩо таке PostgreSQL?, 2021. URL: https://appmaster.io/ru/blog/chto-takoe-postgresql