ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Оптимізація логістичних процесів у сфері вантажних
перевезень за допомогою технології WEB API»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
(код, найменування спеціальності)
освітньо-професійної програми «Інженерія програмного забезпечення»
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

________________ Максим МЕЛЬНИК
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-61
Максим МЕЛЬНИК

Керівник: Наталія ТРІНТІНА
д.т.н., доцент

Рецензент: ________________
науковий ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_______________ Ірина ЗАМРІЙ
«_____» ___________ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Мельнику Максиму Аркадійовичу

1. Тема кваліфікаційної роботи: «Оптимізація логістичних процесів у сфері вантажних перевезень за допомогою WEB API»

керівник кваліфікаційної роботи Наталія ТРІНТІНА д.т.н., доцент,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» жовтня 2023 р. №145.

2. Строк подання кваліфікаційної роботи «29» грудня 2023 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, архітектура WEB API, логістичні процеси, методи отпимізації запитів.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1.Аналіз ключових елементів WEB API.
2.Аналіз існуючих методів та способів отпимізації запитів та логістичних процесів.
3.Аналіз методів та елементів розробки архітектури WEB API.
4. Розробка математичної моделі для визначення найбільш підходящих транспортних засобів.

5. Розробка методу визначення найбільш підходящих транспортних засобів.
6. Розробка WEB API для оптимізації логістичних процесів.

5. Перелік графічного матеріалу: *презентація*

1. Діаграма послідовностей для подання заявки на перевізника замовником через оператора.
2. Діаграма послідовностей для визначення замовником найбільш підходящих транспортних засобів.
3. Визначення критерію якості.
4. Математична модель найбільш вигідних транспортних засобів серед запропонованих.
5. Критерії оцінювання вартості перевезення вантажу.
6. Алгоритм запиту на знаходження оптимальних транспортних засобів.
7. Порівняльний аналіз обробки замовлення користувача.
8. Практичний результат

6. Дата видачі завдання «19» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	19.10-05.11.23	
2	Вивчення матеріалів для аналізу методів оптимізації логістичних процесів	06.11-12.11.23	
3	Дослідження технології WEB API	13.11-19.11.23	
4	Аналіз особливостей використання WEB API у сфері торгівлі	20.11-26.11.23	
5	Розробка математичної моделі для визначення найбільш вигідних транспортних засобів	27.11-03.12.23	
6	Впровадження методу визначення найбільш вигідних транспортних засобів в WEB API	04.12-10.12.23	
7	Оформлення роботи: вступ, висновки, реферат	11.12-20.12.23	
8	Розробка демонстраційних матеріалів	21.12-29.12.23	

Здобувач вищої освіти ______________ Максим МЕЛЬНИК
(підпис)

Керівник
кваліфікаційної роботи ______________ Наталія ТРІНТІНА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 73с., 42 рис., 21 джерел.

Мета роботи – підвищення якості процесу замовлення перевізника.

Об'єкт дослідження – процес замовлення перевізника.

Предмет дослідження – метод визначення найбільш вигідних транспортних засобів серед запропонованих.

У роботі розглянуто способи застосування веб-сервісу у сфері логістики та вантажних перевезень. Виконано огляд методів та алгоритмів для покращення продуктивності, безпеки та масштабованості WEB API. Розглянуто види архітекур при проєктуванні веб-сервісу.

Виконано огляд використаних програмних засобів та середовищ розробки. Розроблено метод визначення найбільш вигідних транспортних засобів серед запропонованих на основі математичної моделі за допомогою мікросервісу, впроваджено метод в WEB API.

КЛЮЧОВІ СЛОВА: МАТЕМАТИЧНА МОДЕЛЬ, WEB API, АРХІТЕКТРУРА, МЕТОД ВИЗНАЧЕННЯ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ.

ABSTRACT

Text part of the master's qualification work: 73 pages, 42 pictures, 21 sources.

The purpose of the work - improve the quality of the carrier ordering process.
Object of research – process of ordering a carrier.

Subject of research – method of determining the most advantageous means of transport among the proposed ones.

In this work, various aspects of using a web service in logistics and cargo transportation are thoroughly analyzed. An overview of different methods and algorithms aimed at improving the performance, ensuring security, and providing scalability of WEB API is carried out. Special attention is given to the consideration of various architectural approaches in designing a web service, allowing for the incorporation of diverse system requirements.

The paper provides an overview of the software tools and development environments used for implementing a web service in the logistics domain. Emphasis is placed on the development of a method for determining the most advantageous means of transport based on a mathematical model. The microservices concept was utilized to implement this method in WEB API.

Key elements of the work include a mathematical model for determining optimal transport solutions, architectural approaches in web service design, implementation of methods to improve the performance and security of WEB API, and an exploration of the software tools used.

The study highlights the significance and relevance of implementing web services in the logistics industry, aimed at improving efficiency and optimizing freight transportation.

The key elements of the WEB API were developed using the C# programming language, and the implemented method was realized through a microservice, which significantly reduced dependencies between systems, thereby increasing the

scalability of the project and facilitating future updates and enhancements. The created service allows for additional interfaces to be developed, introducing solutions in multiple systems through the openness of the WEB API.

KEYWORDS: MATHEMATICAL MODEL, WEB API, ARCHITECTURE, DETERMINATION METHOD, SOFTWARE.

ЗМІСТ

ВСТУП… 11
РОЗДІЛ1 АНАЛІЗ МОДЕЛЕЙ, МЕТОДІВ ТА ЗАСОБІВ СТВОРЕННЯ ВЕБ-СЕРВІСУ 13
1.1 Актуальність WEB API в логістиці 13
1.2 Особливості розробки веб-сервісів 20
1.3 Огляд архітектур для створення WEB API 20
1.3.1 Монолітна архітектура 20
1.3.2 Мікросервісна архітектура...21
1.3.3 Серверна архітектура 22
1.4 Огляд методів та алгоритмів для покращення продуктивності, безпеки та масштабованості WEB API 23
1.4.1 Протоколи та формати обміну даними 23
1.4.2 Оптимізація запитів..25
1.5 Сервіси, інтерфейси, компоненти 28
РОЗДІЛ2 РОЗРОБКА МЕТОДУ ВИЗНАЧЕННЯ НАЙБІЛЬШ ВИГІДНИХ ТРАНСПОРТНИХ ЗАСОБІВ СЕРЕД ЗАПРОПОНОВАНИХ. ВПРОВАДЖЕННЯ ЗАСОБІВ ТА ТЕХНОЛОГІЙ ДЛЯ ОПТИМІЗАЦІЇ ЛОГІСТИЧНИХ ПРОЦЕСІВ 30
2.1 Постановка задачі методу визначення найбільш вигідних транспортних засобів серед запропонованих 30
2.2 Вибір архітектури та способів її впровадження 31
2.3 Застосування синхронності та асинхронності 33
2.4 Прошарки в сервісній архітектурі 36
2.5 Розробка методу пошуку найбільш підходящих транспортних засобів серед запропонованих 44
2.6 Порівняння процесу подання заявки замовником на знаходження перевізника через оператора та WEB API 47
РОЗДІЛ3 ПРОГРАМНА РЕАЛІЗАЦІЯ WEB API ДЛЯ ОПТИМІЗАЦІЇ ВЗАЄМОДІЇ МІЖ ЗАМОВНИКОМ ТА ПЕРЕВІЗНИКОМ 50
3.1 Опис використаних програмних засобів 50
3.1.1 Visual Studio...50
3.1.2 Мова програмування C#...51
3.1.3 Figma 53
3.1.4 Postman...54
3.1.5 Docker...55
3.1.6 GitHub 57
3.1.7 PyCharm 58
3.1.8 Мова програмування Python 59
3.2 Опис класів..60

3.3 Опис методу визначення найбільш вигідних транспортних засобів....73
3.4 Опис бази даних....75
3.5 Приклад застосування 81
ВИСНОВКИ....83
ПЕРЕЛІК ПОСИЛАНЬ....84
ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 86

ВСТУП

У сучасному світі, де логістика відіграє критичну роль у забезпеченні безперервності та оптимізації вантажних перевезень, технології WEB API стають важливим інструментом для покращення ефективності цього процесу. Суттєві зміни в глобальних тенденціях та зростання конкуренції вимагають від суб'єктів логістичного ланцюжка постійного удосконалення та впровадження нових технологій для забезпечення оптимальності управління та реалізації вантажних перевезень.

Важливими показниками ефективності роботи логістики є швидкість та час, з яким оброблюється той, чи інший запит. Наскільки раціонально витрачаються ресурси, як саме забезпечується обмін інформацією між учасниками логістичного процесу, визначає те, наскільки швидко буде доставлятись те, чи інше замовлення, наскільки дорого буде обходитись замовнику доставка самого товару, визначення можливості компанії в кількості одночасно оброблювальних замовлень.

Головне питання, яке ставилось перед створенням веб-сервісу – яким чином оптимізувати процес логістики, а саме комунікацію між замовником та перевізником, відстеження етапу, на якому знаходиться замовлення. Якщо брати спосіб з оператором, головною проблемою є час оброблення одного замовлення та людський фактор, який впливає на те, які перевізники будуть рекомендовані замовнику для перевезення вантажу. Обробка замовлення оператором може займати від 30 хвилин і до декількох робочих днів, в залежності від виду замовлення. При цьому, оператор може навмисно визначити такого перевізника, який буде вигідний саме для нього. Також, є ризик того, що замовлення взагалі не буде оброблене, через те, що він міг про нього забути, або не встигнути в обмежений термін.

WEB API надає можливість розробнику запроваджувати нові реалізації, розширювати та доповнювати проєкт, що дає змогу підлаштовуватись під нові тенденції ринку. Однією із пріоритетних можливостей веб-сервісу полягає в взаємодії між різними компонентами системи, або навіть іншими рішеннями

від сторонніх розробників, які також реалізовані у вигляді API. На такий сервіс можна в будь який момент розробити інтерфейс, за допомогою якого можна зручно та красиво відобразити інформацію на сторінці сайту.

В роботі розглянуто існуючі архітектури, за якими створюється WEB API, застосування веб-сервісів у сфері логістики, особливості розробки, розглянути методи та алгоритми, для покращення продуктивності, безпеки та масштабованості WEB API.

Об'єкт дослідження – процес замовлення перевізника.

Предмет дослідження – технології WEB API для оптимізації взаємодії між замовником та перевізником.

Мета роботи – підвищення якості процесу замовлення перевізника.

Методи дослідження – математичні: статистичні, методи та алгоритми для покращення продуктивності, безпеки та масштабованості WEB API, архітектури розробки: монолітна, сервісна, мікросервісна; методи проєктування та розробки програмного забезпечення, компоненти веб-сервісу.

1 АНАЛІЗ МОДЕЛЕЙ, МЕТОДІВ ТА ЗАСОБІВ СТВОРЕННЯ ВЕБ-СЕРВІСУ

1.1 Актуальність WEB API в логістиці

В сучасному бізнес-середовищі логістика є ключовою галуззю, де WEB API використовується для ефективного управління логістичними процесами. Інтеграція систем дозволяє легко обмінюватися інформацією та уникнення подвійної роботи. Спрощення взаємодії з партнерами через єдину точку доступу полегшує обмін даними.

Відстеження руху товарів в реальному часі та їхня маршрутизація допомагають оптимізувати ланцюг постачань, зменшуючи час доставки та витрати на зберігання. Збір та аналіз великих обсягів даних через WEB API дозволяє приймати обґрунтовані рішення та оптимізувати стратегічний розвиток.

За весь час існування, постійно розвивалися та вдосконалювались технології в логістиці. На початку, всі люди користувались факсом, папером, поштою. Наразі існує потреба в застосуванні більш комплексних систем для аналізу великої кількості даних.

Загалом, можна виділити таку еволюцію: [1]

– 80-і роки. Використання паперу, таксу, телефону та пошти.
– 90-і роки. З'явилися системи підприємств, які надавали послуги застосунків (ASP) (ASP - це підприємства, які надають комп'ютерні послуги клієнтам через мережу).
– 2000-і. Почали стрімко розвиватись та застосовуватись технології, що базуються на хмарах.
– 2010-і. Технологія, яка базується на хмарі продовжує розвиватись. низькі витрати на розробку призвели до поширення нових "легких

технологічних рішень" (SaaS 2.0), призначених для вирішення конкретних проблем ланцюга постачання.

- Останнє десятиліття логістична індустрія пережила вибух нових технологій та інвестицій, до того рівня, де деякі стверджують, що кількість технологічних рішень починає ставати частиною проблеми, оскільки це подальше ускладнює дисперсію та фрагментацію технологій.

Чим далі розвиваються технології, тим більшим стає розуміння того, що потрібно розроблювати системи, які можуть взаємодіяти одна з одною, де кожна виконувала б свою роль та не зачіпляла реалізацію та роботу інших. WEB API є одним із таких рішень, де відбувається “спілкування” між декількома рішеннями, в якому можна все об’єднати в одну цільну картинку. Тепер не буде того, що щоб дізнатися про кількість товару потрібно звертатися в одне місце, для розрахунку вартості або аналізу затримок в інше. Все це має бути уніфіковано, наскільки це дозволяє реалізація проєкта.

Найбільший інтернет магазин у світі Amazon вже у 2006 році запровадив технологію WEB API та визнав її переваги. За допомогою впровадження такої технології, Amazon у 2017 році мав дохід у розмірі 177,9 мільярда доларів.

Інтеграція логістичних систем із електронними магазинами дозволила їм[7]:

- Поліпшити потік інформації між бізнес-суб'єктами.
- Реалізовувати транспортування надійніше
- Скоротити час для впровадження дій, необхідних для задоволення потреб користувачів
- Покращити якість послуг та зменшити загальні витрати, що виникають внаслідок спільного утримання діяльності
- Проводити складні аналізи (які користувачі найчастіше використовують які послуги та платформи).

В роботі [8] автором описано весь етап створення проєкту для модернізації системи управління замовленнями K-Con, надаючи

централізований доступ до даних про замовлення на настільних комп'ютерах та мобільних пристроїв. В ній описано, що виникала проблема з непорозумінням між замовником(в даному випадки агенти компанії K-Con) та водіями вантажівок, коли всі дані зберігались в одному файлі і були конфлікти зі злиттям файлу. Потрібно було постійно координуватися з операторами, щоб уникнути конфліктів. У вдосконаленій ситуації дані про замовлення зберігаються в СКБД на централізованому сервері. Дані про замовлення можна отримати за допомогою RESTful веб-сервісу. Веб-додаток із оптимізованими переглядами для мобільних пристроїв дозволяє агентам працювати поза офісом. Одночасний доступ для читання та запису до даних про замовлення є можливим. Огляд цієї роботи є достатньо важливим, адже дає розуміння того, що можна вдосконалити в процесі логістики. Як ми бачимо, часто виникають непорозуміння між учасниками системи. Можливе рішення – це налагодження процесу комунікації та відстеження замовлень, яке допоможе учасникам легше орієнтуватися.

Після аналізу інтерфейсів та їх структур даних виявляється, що більшість користувачів обмежується форматом даних (файлу) json або xml, причому файл json або xml може мати зовсім різні структури даних та назви елементів даних. Це призводить до створення нових API-інтерфейсів для того самого бізнес-процесу з іншим бізнес-партнером (наприклад, іншим LSP). Таким чином, виглядає, що існує велика область для узгодження API-інтерфейсів, найкраще на рівні референс-моделі даних, яка була б технологічно агностичною і може бути використана як для API REST-json, так і для SOAP - xml, і фактично не тільки для API. Слід відзначити, що ми часто маємо справу з інтерфейсами API, які дозволяють передавати людиночитабельні документи, такі як PDF або графічні формати. Ці документи - це етикетки, рахунки, накладні тощо. Це через те, що ці документи все ще потрібні через законодавчі норми, аудити або тому, що процеси ще не були повністю адаптовані до ефективності, яку пропонує EDI або API.

Як приклад цікавого інтерфейсу можна згадати викиди CO2 або геолокацію. Іншою річчю, яка характеризує інтерфейси API, особливо ті, які є на платформах, є інтерфейси, які призначені тільки для використання комп'ютерними програмами. Їхня функція полягає в отриманні інформації, необхідної для виконання подальших кроків у роботі програми, що, звісно ж, відповідає дусі API, де зовнішні дані можна обробляти як місцеві ресурси[9].

В роботі [10] описується робота електронних комерцій під час пандемії COVID – 19. Тоді логістика відігравала важливу роль в забезпеченні можливості бізнесам надавати основні послуги, гарантуючи, що товари, куплені споживачами в інтернет-магазинах, надходять до них своєчасно. Платформи електронної комерції надали продавцям можливість швидко адаптуватися та легко налаштовувати онлайн-бізнеси. Це також дало продавцям гнучкість інтегрувати послуги 3PL, які відповідають їх бізнес-потребам, щоб вони не обмежувалися лише тим, що пропонують платформи електронної комерції.

API логістики стали більшою необхідністю на сучасному ринку, тому їх захист - це обов'язково. Його ігнорування викладає користувачів ризикам витоку даних, що дозволяє зловмисникам розробляти більше атак, які можуть завдати більше шкоди. Пам'ятайте, що ці недоліки безпеки не є складними для зловмисників, тому навіть кіберзлочинець низького рівня може їх використовувати. Для уникнення таких ризиків забов'язані ключові учасники поведінки в області кібербезпеки. Запровадження двохфакторної аутентифікації, налаштування токенів та ролей може значно підвищити захист додатків та даних клієнтів.

API дозволяє зовнішнім розробникам підключатися до бізнес-активів постачальника API та створювати нові застосунки на основі цих активів. Речі, такі як дизайн, технічна якість та якість документації, значно впливають на те, наскільки легким є цей процес. API є важливим будівельним блоком, якщо організація хоче перетворити свій продукт на основі веб-сервісів чи мобільний продукт в платформу. Так званий гравітаційний центр платформи в значній мірі залежить від характеру бізнес-активів. Якщо активи є цінними і корисними для

достатньо великої групи користувачів, перспективи платформи є відмінними. Постачальник API також може придбати або найняти успішних розробників додатків сторонніх сторін[11].

В роботі [12] описуються випадки із відомих компаній, такі як Amazon, Netflix, які стикнулися з труднощами в кодуванні та міжзалежностями в управлінні. Вони описували свої проєкти як один великий архітектурний кам’яний пам’ятник, у якій всі функції були дуже тісно пов’язані одна з одним. Але з часом, коли їх клієнтська база почала розростатися, коли додаються все більше розробників, кодова база стає все більшою і система вже потребує масштабування та декомпозиції. Ось як це зробив Amazon: розробники проаналізували вихідний код і виокремили блоки коду, які виконували одну функціональну мету.

Вони оточили ці блоки інтерфейсом веб-сервісу. Наприклад, вони розробили один сервіс для кнопки "Купити" на сторінці товару, один сервіс для функції калькулятора податку і так далі. Amazon призначив відповідальність за незалежний сервіс команді розробників. Це дозволило командам бачити більше деталей щодо заторів у розвитку і вирішувати труднощі більш ефективно, оскільки кілька розробників могли зосередити свою увагу на одному конкретному сервісі. Щодо підключення мікрослужб для формування більшого додатка: вирішення проблеми роботи для однієї функції було створенням стандарту, якому повинні були слідувати розробники, що функції могли спілкуватися лише за допомогою своїх власних API веб-служб. Мікрослужби розвинулися, щоб стати більш вдалими для складних застосувань і є передовим рішенням, коли постійне вдосконалення та розвиток веб-сайтів і послуг є стандартом.

У цьому дослідженні [13] SCM відіграє ключову роль у зв'язуванні етапів у всьому процесі, включаючи клієнтів та постачальників. Прототип було розроблено на основі об'єктно-орієнтованого моделювання за допомогою Unified Modeling Language (UML). Основні сутності, необхідні для симуляції ланцюга постачання, представлені через UML діаграми використання та

діаграми послідовності для ілюстрації взаємозв'язків та взаємодій між цими сутностями і для опису потоку дій у всьому процесі. Результати показують, що фреймворк може легко моделювати прості обставини. Цей простий сценарій можна легко розширити для включення кількох продуктів та багатоетапів.

SCM: Управління ланцюгом постачання на основі ЕС та ЕДІ інтегровано в засіб проектування на основі SCM для підтримки діяльності проектування. Запропонована система може скоротити час обробки в транспорті та виробництві. Цей засіб проєктування на основі SCM можна розглядати як один з рішень у проєктуванні для подолання обмежень.

У цій роботі[14] було продемонстровано застосування Контекст-Орієнтованості в рамках постачання матеріалів у процесі будівельно-логістичних дій. Сценарій проекту показав, що концептуально Контекст-Орієнтованість може бути інтегрована з Веб-сервісами з метою забезпечення надання відповідної інформації та підвищення співпраці в ланцюгу постачання будівельно-логістичної сфери.

Вона стверджує, що паралельні розробки у сфері Веб-сервісів, які забезпечують здатність динамічного виявлення та виклику сервісів незалежно від операційної системи чи мови програмування, можна використовувати для надання членам будівельного проекту можливості доступу в реальному часі до різних корпоративних систем та ресурсів кількох міжпідприємницьких проектів. Інтеграція Контекст-Орієнтованості та Веб-сервісів у систему управління будівельно-логістичним ланцюгом постачання має значний потенціал для покращення логістичних послуг і забезпечення доступу кожної послуги до контекстно-специфічних даних, інформації та сервісів. Впровадження IWW в будівельно-логістичну сферу може покращити операцію JIT, зменшити відходи і збільшити міжфункціональні дії. Крім того, запропоновану нову бізнес-модель можна використовувати як посилання для майбутньої будівельної логістики, де логістичні послуги можуть бути класифіковані як один із важливих блоків в загальній мережі будівельного ланцюга постачання.

Автори роботи [2] запропонували P2P-структуру, яка дозволить підтримувати взаємодію між різними учасниками логістичних ланцюгів, безпечно обмінюючи даними між різними сторонами та використовуючи їх для оптимального планування на основі даних IoT. Розробники вирішили спільно перейти до загальної логістичної інтеграції на основі відкритих інтерфейсів програмування застосунків (API). Виділити з цього можна впровадження ними інтеграції логістичних API з Blockchain та смарт-контрактами для обміну цифровою інформацією та інтеграції смарт-контрактів для управління автоматизації транзакцій.

Таким чином, вони досягли наступного:

- Реальний час видимості транспортування логістики: моніторинг та контроль логістичних процесів дозволяють розробку алгоритмів прогнозування попиту та постачання на основі моделі витягу для управління доставкою з метою скорочення часу доставки, часу очікування та ручної роботи.
- Забезпечення збереження та обміну даними за допомогою Blockchain для транзакцій B2B та IoT.
- Автоматизація інтеграції постачання та логістики в загальному організаційному середовищі.

В роботі [3] автори стикнулися з проблемою з балансуванням кількох архітектур і технологій із консолідацією великої кількості даних для створення інтерактивного візуального досвіду. Для вирішення цього виклику система була розділена на 4 різні підсистеми. Кожна підсистема розроблена для підтримки використання різних технологій та/або мов програмування. Підсистеми "спілкуються" одна з одною за допомогою реалізаційно-нейтрального API. Це дозволяє використовувати найкращий інструмент для кожного компонента, полегшуючи можливість зміни підсистем (для покращення продуктивності) без негативного впливу на інші частини додатку.

1.2 Особливості розробки веб-сервісів

Розробка веб-сервісів є складним процесом, який вимагає від розробників розуміння ряду ключових аспектів. Важливо враховувати архітектурні принципи, такі як REST або SOAP, для забезпечення ефективної комунікації між клієнтами та серверами. Розробники повинні уважно обирати технології, мови програмування та фреймворки, враховуючи вимоги проєкту та його масштабованість. Суттєвим етапом є забезпечення безпеки веб-сервісів, включаючи аутентифікацію, авторизацію та захист від атак.

Важливо розуміти принципи обробки та збереження даних, використовуючи відповідні бази даних та механізми збереження інформації. Тестування, зокрема юніт-тестування та інтеграційне тестування, є необхідним елементом розробки веб-сервісів для забезпечення надійності та виявлення помилок на ранніх етапах. Крім того, розробники повинні удосконалювати документацію для забезпечення зрозумілості та легкості використання веб-сервісів іншими розробниками. Налаштування на принципи мікросервісної архітектури може полегшити управління та розгортанням веб-сервісів. Загалом, розробка веб-сервісів вимагає глибокого розуміння технічних аспектів, врахування вимог проєкту та прагнення до створення високоякісного, ефективного та безпечного програмного забезпечення.

Різниця у використанні різних архітектур полягає не тільки у тому, як саме буде будуватися веб-сервіс, але й взаємодія між різними частинами коду.

1.3 Огляд архітектур для створення WEB API

1.3.1 Монолітна архітектура

Монолітна архітектура є простою та ефективною для невеликих та середніх проєктів. За рахунок того, що структура цієї архітектури побудована таким чином, що всі частини взаємодіють між собою, без зовнішніх інтерфейсів чи мережевих викликів, вона є неефективною для веб-сервісів, які потребують

співпрацю з іншими реалізаціями, інтеграції. Масштабованість при такому підході помітно знижується, адже доведеться підлаштовувати реалізацію під вже написаний код. Через свою зав'язність між частинами коду, додавання нових функцій буде займати велику кількість часу. На відміну від тієї ж серверної архітектури, використання мікросервісів унеможливлюється та все зосереджується на тому, як розширити вже існуючу логіку. Покриття інтеграційними тестами буде важчим. З іншого боку, така архітектура має свою перевагу в простоті розгортання самого проєкту. Весь програмний продукт розгортається однією одиницею(контейнер, сервер). Управління та ведення розробки такого проєкту легше для однієї великої команди, але виникають проблеми при залучені сторонніх розробників.

1.3.2 Мікросервісна архітектура

Мікросервісна архітектура пропонує підхід до розробки програмного забезпечення, в якому програмний продукт(додаток) розбивається на невеликі незалежні сервіси, які взаємодіють між собою через мережу. Кожен мікросервіс є окремим функціональним компонентом, розгортаним і відновлюваним незалежно від інших. Завдяки цьому, кожен сервіс є окремим функціональним компонентом, здатним функціонувати та розгортатися незалежно від інших. Всі сервіси спілкуються між собою через чітко визначені API(Application Programming Interface), які визначають яким чином сервіси можуть комунікувати між собою через певні визначені контракти.

Також, через це, кожен сервіс має свою зону відповідальності. Вони відповідають за конкретну функціональність або реалізацію, через що це сприяє розподіленню відповідальностей, описану вище, та покращує модульність системи.

Такі сервіси можуть бути розгорнуті автономно. Масштабування кожного такого сервісу виконується окремо. Якщо є можливість, над окремим із них можуть працювати власні розробники, залучуючи власні ресурси. Зміни в них не будуть впливати на зміни в іншому, але зі своїми нюансами. Якщо один

сервіс використовується в іншому, тобто, йде його виклик, від якого очікується певний результат, зміна реалізації одного мікросервіса може видати результат, який інший не очікував. Тобто, реалізація одного не вплине на реалізацію іншого напряму, але, через зміну логіки може змінитися, наприклад, результат, який мав повертати мікросервіс. Якщо це не врахувати, або навіть не знати про це, можуть виникнути неочікувані помилки, коли від такого мікросервісу очікували одну відповідь, а видалась інша. Такі зміни мають обов'язково бути обговорені між розробниками.

Через свою автономність, різні мікросервіси можуть використовувати різні технології та мови програмування в залежності від їхньої конкретної задачі та потреб. Там, де в одній мові програмування відсутні необхідні бібліотеки, в іншій їх може бути декілька. Можна враховувати переваги мови програмування, якщо питання ставиться в оптимізації часу роботи або в певній специфіці, як приклад вибір мови за її типізацією(динамічна або статична), від чого залежать її сильні сторони.

1.3.3 Серверна архітектура

Серверна архітектура - це структура та організація серверів, які використовуються для обробки запитів та забезпечення функціональності в розподіленій системі. Ця архітектура визначає, як різні сервери взаємодіють один з одним та як вони служать клієнтам. Модель взаємодії може бути синхронною, коли клієнт чекає на відповідь, або асинхронною, коли використовуються повідомлення. Важливо розрізняти різні типи серверів, такі як веб-сервери, додаткові сервіси та проксі-сервери, кожен із яких виконує свої функції у розподіленій системі.

Різниця між типами серверів. Веб-сервери частіше оброблюють HTTP-запити, відповідаючи на них, надсилаючи при цьому або статичні або динамічні веб-сторінки.

Додаткові сервіси надають додатковий функціонал. Як приклад, бази даних, кешування, розподілені обчислення.

Проксі-сервери зазвичай використовуються для перенаправлення запитів, коли потрібно з однієї сторінки перекинути на іншу. Також, такі сервери використовують для балансування навантаження, коли система помічає, що навантаження на один сервер перевищує норму очікування відповіді. Тоді, застосовується балансери, які застосовують реплікацію серверу.

Методи взаємодії між серверами можуть бути реалізовані через HTTP/RESTful API, RPC або через використання черг повідомлень для асинхронної комунікації. Балансування навантаження використовується для оптимізації продуктивності серверів, існує можливість горизонтального та вертикального масштабування. Кешування, резервне копіювання та відновлення, аутентифікація та авторизація, SSL/TLS-шифрування - це елементи, що забезпечують безпеку та доступність системи.

У випадку використання серверної архітектури важливо враховувати не лише технічні аспекти, але й бізнес-потреби. Ця архітектура допомагає забезпечити ефективність, масштабованість та надійність системи. Основний вибір між синхронною та асинхронною моделями взаємодії визначається вимогами до продуктивності та часу відгуку системи на запити клієнтів.

Також важливо розглядати сучасні підходи до розробки, такі як мікросервісна архітектура, яка передбачає розділення функціональності на невеликі, незалежні сервіси, що полегшує розробку та підтримку системи. Використання відкритих стандартів, таких як HTTP та REST, дозволяє покращити інтеграцію та взаємодію між різними компонентами системи.

1.4 Огляд методів та алгоритмів для покращення продуктивності, безпеки та масштабованості WEB API

1.4.1 Протоколи та формати обміну даними

Протоколи та формати обміну даними визначають спосіб, за яким інформація передається між клієнтом та сервером у веб-сервісах. Два основні елементи у цьому контексті - протоколи транспорту та формати даних.

Протоколи використовуються для передачі даних через мережу. Інсують два протоколи передачі даних - HTTP (Hypertext Transfer Protocol) та HTTPS (Hypertext Transfer Protocol Secure). Наразі, HTTP майже не застосовується, адже існує більш модифікований HTTPS, який має приставку Secure.

HTTP використовується для передачі різноманітних запитів (GET, POST, PUT, DELETE) та отримання відповідей. GET використовується для отримання даних, POST - для відправки даних на сервер, PUT - для модифікації існуючих даних, а DELETE - для видалення. Основна мета протоколу HTTPS є забезпечення конфіденційності, цілісності та автентифікацію даних під час передачі даних.

Протокол використовує протоколи шифрування, такі, як SSL (Secure Sockets Layer) або його сучасний нащадок, TLS (Transport Layer Security), для захисту даних від проміжних атак і перехоплення. Це дозволяє забезпечити конфіденційність інформації під час передачі.

Формати даних надають спосіб подання даних у вигляді певної структури, що передається між клієнтом та сервером. Одними із найпоширеніших форматів є JSON(JavaScript Object Notation) та XML(eXtensible Markup Language). JSON представляє собою формат обміну даними, який відрізняється своєю простотою та легкістю розуміння. Його основні елементи - це рядки, числа, булеві значення, а також структурні об'єкти та масиви. JSON отримав широку популярність в сфері веб-сервісів завдяки своїй зручності для читання, зрозуміння та легкості обробки програмами.

XML, навпаки, є більш стандартизованим та розширюваним форматом для представлення структурованих даних. Він використовує теги та атрибути для визначення структури документа. Незважаючи на свою широку застосовність в різних галузях, порівняно з JSON, він відзначається більшим обсягом та складнішим процесом обробки.

1.4.2 Оптимізація запитів

Оптимізація запитів є важливим аспектом при розробці WEB API. За рахунок впровадження таких процесів, збільшується швидкодія обробки HTTP-запитів, які в свою чергу дозволяють обробити більше потенційних клієнтів.

Один із процесів оптимізації запитів є кешування. Операція кешування - це процес зберігання раніше отриманих даних для уникнення витрат при повторних запитаннях з аналогічним або тотожнім змістом. Застосування кешу підвищує продуктивність веб-сервісу, зменшує навантаження на сервер і покращує відгук системи. Цей процес ґрунтується на тимчасовому зберіганні інформації на локальному рівні, що дозволяє швидко доступатися до неї без необхідності повторного запитання на сервері. З використанням кешу можна досягти ефективнішого використання ресурсів та скорочення часу відповіді на запити, оскільки часто використовувані дані вже знаходяться в готовому стані на місці їхнього використання.

Застосування асинхронного та паралельного програмування стає ключовим елементом оптимізації обробки великої кількості запитів одночасно. Асинхронність дозволяє серверу неперервно виконувати обробку інших запитів, навіть під час очікування завершення конкретних операцій. З іншого боку, паралельність використовує різні ресурси для симультанного опрацювання декількох запитів, що призводить до більш ефективного використання обчислювальних потужностей та прискорення загального часу виконання. Ці дві стратегії в сукупності роблять можливим ефективне та швидке виконання різноманітних завдань у вимогливих до ресурсів середовищах.

При розробці веб-сервісу виникає необхідність ретельніше розглядати стратегії мінімізації обсягу передаваних даних між клієнтом та сервером. Це не лише сприяє зменшенню часу передачі, але й оптимізує використання пропускної здатності мережі, важливе завдання в умовах сучасного інтернет-екосистеми.

У цьому контексті мінімізація включає в себе застосування різноманітних підходів, таких як стиснення даних та використання компактних форматів передачі. Один із підходів може полягати в ухваленні оптимальних стратегій стиснення, що сприяють збереженню якості інформації при одночасному скороченні її обсягу. Переваги використання компактних форматів, таких як JSON, у порівнянні з більш об'ємними форматами, наприклад XML, слід ретельно аналізувати в контексті конкретних потреб системи.

Крім того, ефективна мінімізація передбачає відправку лише тієї інформації, яка необхідна для виконання конкретного завдання. Цей підхід може включати в себе вдосконалені методи визначення та фільтрування зайвих даних, забезпечуючи точність та раціональність обміну інформацією між системними компонентами.

Буферизація, як принцип, дозволяє ефективно керувати передачею даних, збираючи їх у певний час в пакети перед їхньою відправкою. Цей метод сприяє зменшенню кількості окремих запитів, оптимізуючи взаємодію між клієнтом та сервером. При цьому важливо досліджувати оптимальні параметри буферизації для досягнення максимальної продуктивності та мінімізації затримок в обміні даними.

Пакетування даних, в свою чергу, є важливою стратегією для оптимізації передачі великих обсягів інформації. Цей процес дозволяє передавати значущі об'єми даних за один раз, підвищуючи тим самим швидкість та ефективність комунікації. В роботі важливо розглядати різні методи пакетування, враховуючи їхню придатність до конкретних умов взаємодії.

Комбінування буферизації та пакетування дозволяє створити високоефективний механізм оптимізації передачі даних у системах зв'язку.

SQL – запити це ключовий елемент, який використовується в базах даних. Оптимізація SQL-запитів включає в себе ряд стратегій, спрямованих на максимальне використання ресурсів бази даних. Перш за все, це використання індексів, що дозволяє значно прискорити виконання запитів шляхом швидкого

доступу до необхідних даних. Важливо проводити детальний аналіз структури бази даних з метою її оптимізації, враховуючи особливості та потреби конкретного веб-сервісу.

Додатково, важливим етапом є вибір оптимальних SQL-запитів, орієнтованих на мінімізацію часу виконання та оптимізацію використання ресурсів сервера бази даних. Ретельне тестування та вдосконалення запитів стає необхідним етапом оптимізації, спрямованою на підвищення ефективності системи.

Не варто забувати і про інструменти моніторингу та профілювання з метою виявлення та усунення слабких місць у функціонуванні веб-сервісів. Моніторинг, як процес, надає можливість постійно відслідковувати ефективність веб-сервісу в режимі реального часу. Це дозволяє оперативно виявляти та реагувати на потенційні проблеми, що можуть виникнути під час експлуатації. Аналіз метрик продуктивності дозволяє вчасно вживати заходів для оптимізації роботи системи та запобігання можливим перебоям.

З іншого боку, профілювання виявляється важливим інструментом для ідентифікації частин коду, які вимагають удосконалення. Аналіз викликів функцій та вимірювання часу їх виконання дозволяє точно визначити та виправити проблемні аспекти в програмному забезпеченні. Ефективне використання профілювання сприяє оптимізації кодової бази та покращенню продуктивності.

В сукупності, використання стратегій моніторингу та профілювання є необхідною складовою оптимізації веб-сервісів, спрямованою на забезпечення їхньої стабільності, ефективності та високої якості обслуговування користувачів.

CDN(Content delivery network) – система взаємопов'язаних серверів, розташованих у різних географічних областях та пов'язаних мережами для оптимізації доставки контенту до кінцевих користувачів. Цей механізм сприяє скороченню часу завантаження ресурсів для кінцевого користувача завдяки

оптимізації шляху передачі даних та мінімізації мережевих затримок. Динамічне розташування контенту на серверах, найближчих до користувача місцевостей, покращує не лише швидкість завантаження, але й загальну продуктивність веб-сервісу.

Окрім цього, CDN може виявитися важливим інструментом для оптимізації обробки трафіку та зниження навантаження на центральний сервер. Розподілення навантаження між різними вузлами мережі дозволяє оптимально використовувати ресурси та підтримувати стабільну роботу веб-сервісу навіть під час пікових навантажень.

Узагальнюючи, детальне розглядання та ефективне впровадження CDN представляє собою важливий етап оптимізації веб-сервісів, спрямований на покращення доступності та продуктивності для користувачів у різних частинах світу.

1.5 Сервіси, інтерфейси, компоненти

Один з ключових аспектів послуг полягає у розрізненні між інтерфейсною та реалізаційною частинами. Інтерфейсна частина визначає функціональність, яка доступна зовнішньому світові, і засоби доступу до цієї функціональності. Сервіс описує власні характеристики інтерфейсу, такі як доступні операції, параметри, типи даних та протоколи доступу. Це робиться так, щоб інші програмні модулі могли з'ясувати, що робить сервіс, як викликати його функціональність і який результат очікувати. Сервіси, у цьому контексті, є програмними модулями, які укладають контракт, надаючи публічно доступні описи інтерфейсних характеристик для потенційних клієнтів.

Реалізація, у свою чергу, втілює цей інтерфейс, і деталі реалізації залишаються прихованими від користувачів сервісу. Різні постачальники сервісів можуть реалізувати однаковий інтерфейс, використовуючи будь-яку обрану мову програмування. Наприклад, одна реалізація сервісу може надавати

функціональність безпосередньо, тоді як інша може використовувати комбінацію інших сервісів для забезпечення тієї ж функціональності.

Остаточною метою обчислення, орієнтованого на сервіси, є створення розподілених додатків, які можуть динамічно збиратися відповідно до змін бізнес-потреб і налаштовуватися відповідно до доступу пристроїв і користувачів.

Щоб краще зрозуміти, як проєктувати та розробляти сервіси, важливо розглянути взаємозв'язок між сервісами, інтерфейсами та компонентами. Під час розробки додатка розробники створюють логічну модель того, які бізнес-об'єкти використовує підприємство (наприклад, продукт, клієнт, замовлення) та які сервіси вони потребують від цих об'єктів. Розробники можуть реалізувати ці концепції через комбінацію інтерфейсних специфікацій та компонентних реалізацій, які використовуються для втілення функціональності сервісу.

Сервіси мають бути спроєктовані та реалізовані так, щоб їх можна було повторно використовувати в різних контекстах, які визначають споживачі сервісу. Це схоже на "людські послуги", де, наприклад, не має значення, що міститься в конверті, вартість відправлення залишається постійною, незалежно від його вмісту. Також важливо відрізняти інтерфейси сервісів від реалізаційних компонентів, оскільки часто постачальники інтерфейсів не тотожні із постачальниками самого сервісу. Сам сервіс є бізнес-концепцією, що визначається з урахуванням застосунків або користувачів, тоді як його втілення може бути реалізоване різними програмними пакетами чи застосунками[21].

2 РОЗРОБКА МЕТОДУ ВИЗНАЧЕННЯ НАЙБІЛЬШ ВИГІДНИХ ТРАНСПОРТНИХ ЗАСОБІВ СЕРЕД ЗАПРОПОНОВАНИХ. ВПРОВАДЖЕННЯ ЗАСОБІВ ТА ТЕХНОЛОГІЙ ДЛЯ ОПТИМІЗАЦІЇ ЛОГІСТИЧНИХ ПРОЦЕСІВ

2.1 Постановка задачі методу визначення найбільш вигідних транспортних засобів серед запропонованих

Коли замовник залишає своє замовлення на перевозку певного товару, він завжди має зачекати певний час, поки оператор підбере відповідного перевізника для нього. Цей процес також є нешвидким, оператор має порівняти відповідні характеристики для того, аби зрозуміти чи дійсно той, чи інший перевізник дійсно підходить для перевезення цього вантажу. Після цього йде процес комунікації між оператором та перевізником, який теж займає певний час. Після всіх обговорень, оператор, нарешті, може сповістити замовника про знайденого для нього перевізника. Весь цей процес може тривати від 30 хвилин і більше, закінчуючи декількома робочими днями. Все це створює пробіли та затримки в логістичному процесі.

Щоб позбутися таких затримок та оптимізувати процес визначення перевізника для замовника, створено метод, який буде в цьому допомагати. Перевага розробленої системи – WEB API. Через те, що в системі виключений такий учасника, як оператор, спростився сам процес оформлення замовлення та пошуку відповідного перевізника. Тим паче у випадку з оператором досить сильно на весь процес впливає людський фактор та досвід самого оператора. Чим досвідченіший оператор, тим швидше він знайде підходящу кандидатуру. Але, оператор може бути неправдивим і обирати перевізника, який за певних причин вигідний для оператора. Можливо він з цього отримає певний відсоток до заробітньої плати, або інші подібні переваги. Сам метод визначення таких перевізників, виключає подібні недоліки і за допомогою переваги технології

WEB API робить цей процес більш уніфікованим, зручнішим та швидшим. Метод дозволить нам не тільки визначити перевізника, але й дасть нам суму, в яку обійдеться доставка такого замовлення.

Також, через те, що використовується відображення інформації на сторінках, системі нічого не буде вартувати відобразити відповідну інформацію про вантажний засіб самого перевізника, необхідну інформацію про нього, таку, як номер телефону, ініціали, всі контактні дані. Окрім цього, перевізників може бути скільки завгодно, а отже є вибір кого саме можна обрати. Якщо у випадку з оператором, йому довелось би опитувати кожного такого перевізника, у WEB API перевізники самі залишають свою заявку на участі в перевезені певного вантажу. Таким чином, формується список з таких перевізників, де можна переглянути інформацію по кожному із них. У замовника тепер є можливість самостійно обирати вигідного для нього перевізника, або ж скористатися методом, який самостійно вкаже йому, кого краще взяти для перевезень вантажу.

2.2 Вибір архітектури та способів її впровадження

Першою причиною вибору архітектури для її впровадження в систему це потреба у масштабуванні, доповненні та підтримці. В розробленій системі, всі перечисленні потреби є головними та варто обирати ту архітектуру, яка змогла б вирішити їх всі, або хоча б багато з них.

Якщо розглядати монолітну архітектуру, тут виникає багато проблем з її недоліками, які заключаються в тому, що її необхідно підтримувати однією командою розробників та вона дуже важко піддається розширенню. Тим паче, що запровадження сторонніх рішень в такій системі буде дуже складною задачею. Потрібно буде або перероблювати вже існуюче рішень під потреби проєкту та вносити його всередину існуючих рішень, або домовлюватись з розробниками для запровадження додаткової логіки, щоб стороннє рішення працювало з тим, куди його хочуть запровадити. Обидва варіанти займуть дуже

багато часу і не факт, що це принесе тих результатів, які планувались. Рішення може з часом виявитись непридатним для подальшого використання через свою неактуальність, і тоді доведеться вже або переписувати його, або взагалі вирізати з проєкту.

Серверна архітектура вирішує більшість з них. Використання такої архітектури дозволить нам визначати, як один сервіс може взаємодіяти з користувачем. В такому випадку, весь проєкт розбитий на окремі шари, кожен з який може взаємодіяти з іншими за певними прописаними правилами. Всі ці правила встановлюють самі розробники, і налаштовуються відповідно до вимог проєкту. В розробленій системі будуть присутні чотири шари, кожен з яких буде спілкуватися з іншими через контракти. Без цих контрактів доступ буде заблокований. Контракти по своїй суті це опис наявних методів, який має той, чи інший клас, реалізація якого прихована. Для проєкту така архітектура підходить та допоможе вирішити багато проблем. З її застосуванням у нас є можливість залучати у разі потреби інших розробників, які не мають вникати повністю у всю систему, як це було б з монолітною архітектурою, де щоб запровадити якусь зміну, потрібно розуміти взаємозв’язки між іншими.

Серверна архітектура дає можливість залучати розробників для роботи над окремими шарами без необхідності вивчення всієї системи. Зв'язність об'єктів (сервісів) зменшується, що дозволяє розробникам зосередитись та змінювати конкретні частини системи, не впливаючи на інші. Також, серверна архітектура сприяє більш гнучкому розширенню чи зміні окремих компонентів, не впливаючи на основний проєкт в цілому.

Щоб застосувати таку архітектуру, потрібно розбити її на відповідні прошарки, кожен з яких буде відповідати певній функціональності в системі. Через те, що було обрано серверну архітектуру, з’явилась можливість впроваджувати і мікросервіси, за домогою якого буде розраховуватись найбільш підходящі транспортні засоби для замовника. При цьому, не буде потреба в рефакторингу всього коду, а лише її відповідної частини, прошарку.

2.3 Застосування синхронності та асинхронності

Синхронні послуги викликають клієнти, висловлюючи своє прохання у вигляді виклику методу з набором параметрів, який повертає відповідь із значенням. Це означає, що, коли клієнт висилає запит, він очікує відповіді, перш ніж продовжити свій розрахунок. Завдяки такому двосторонньому зв'язку між клієнтом та сервісом, сервіси у стилі RPC потребують тісно зв'язаної моделі спілкування.

Їх зазвичай використовують, коли додаток відповідає наступним характеристикам:

– Клієнт, який викликає сервіс, очікує негайної відповіді.
– Клієнт та сервіс взаємодіють у спосіб зворотнього спілкування.
– Сервіс орієнтований на процес, а не на дані. Наприклад, повернення поточної ціни на акції, надання поточних погодних умов або перевірка кредитного рейтингу можливого торговельного партнера перед завершенням угоди.

Асинхронні послуги є послугами у стилі документу або орієнтованими на повідомлення. Коли клієнт викликає таку послугу, він, як правило, надсилає весь документ, такий як замовлення на покупку, замість окремого набору параметрів. Сервіс приймає та обробляє весь документ, можливо, повертаючи або не повертаючи результат. Клієнт, який викликає асинхронну послугу, може продовжити виконання свого застосунку без очікування відповіді. Відповідь може з'явитися через години або навіть дні. Такий тип послуг сприяє меншому зв'язку між клієнтом та постачальником, оскільки не вимагає тісно зв'язаної моделі запит-відповідь.

Застосування асинхронності доцільне у наступних випадках:

– Клієнт не очікує негайної відповіді.
– Сервіс орієнтований на процес. Наприклад, обробка замовлення на покупку, відповідь на запит на цінову пропозицію від клієнта чи реакція на розміщення замовлення певним клієнтом. Клієнт надсилає

веб-сервісу цілий документ та вважає, що веб-сервіс обробляє його, але не потребує негайної відповіді.

В проєкті було застосовано асинхронність для того, аби була можливість обробляти одночасно велику кількість даних. Так, як було використано технологію WEB API, відповідно одну й ту саму операцію можуть виконувати одночасно сотні, а й можливо навіть тисячі користувачів. Щоб кожен користувач не чекав своєї відповіді дуже довгий час, а таких користувачів, знову ж таки, одночасно може бути дуже багато, кожен такий запит буде виконуватись асинхронно.

```
[HttpPost("register")]
[AllowAnonymous]
[ProducesResponseType(typeof(UserCreatedResponse), StatusCodes.Status201Created)]
[ProducesResponseType(typeof(ApiErrorResponse), StatusCodes.Status400BadRequest)]
Ссылок: 0
public async Task<IActionResult> RegisterAppUser([FromBody] UserRegisterRequest request)
{
    await _authValidatorsAggregate.UserRegisterValidator.ValidateAndThrowAsync(request);
    var userCreatedResponse = await _authService.CreateUser(request);

    return StatusCode(StatusCodes.Status201Created, userCreatedResponse);
}
```

Рис. 2.1 Приклад застосування асинхронності в роботі в сервісі реєстрації користувача

На цьому зображені видно, що сам метод використовує асинхронність, викликаючи сервіс створення користувача(CreateUser). Таким чином, було реалізувано асинхронність через використання асинхронного методу в програмному коді. За рахунок асинхронності, є можливість продовжувати виконувати програмний код, не очікуючи відповідь від методу. Теж саме можна сказати і про операції, які можуть зайняти час (наприклад, робота з базою даних), виконуються асинхронно, не блокуючи викликаючий потік виконання. В даному проєкті при роботі з базою даних, асинхронність присутня повсюди, але потрібно бути уважним і слідкувати за тим, щоб передбачити помилки, які можуть виникнути в методі, адже можуть виникнути помилки під час

виконання асинхронної операції, або при виклику результатів цієї операції. Якщо правильно виконати обробку методу, цього можна уникнути.

```
Ссылок: 1
public async Task CommitTransactionAsync(Action action)
{
    await using var transaction = await _databaseContext.Database.BeginTransactionAsync();

    try
    {
        action();
        await _databaseContext.SaveChangesAsync();
        await transaction.CommitAsync();
    }
    catch (Exception exception)
    {
        _logger.LogError(exception, "Exception occured in transaction: {Message}", exception.Message);
        await transaction.RollbackAsync();
        throw;
    }
}
```

Рис. 2.2 Приклад обробки методу

Всі внутрішні виклики в даному методі, який відповідає за запис транзакції, використовує конструкцію try catch. Ця конструкція використовується для обробки винятків, тобто ситуацій, коли відбувається помилка в ході виконання програмного коду. Вона дозволяє "спробувати" виконати певний код і перехопити будь-які винятки, якщо вони виникають, для того, щоб виконати альтернативні дії або повідомити про помилку. В цьому випадку, перехоплюється помилка, яка має тип Exception(системний тип). Далі йде спроба зберігання даних в базі. У випадку, якщо ця операція буде неуспішною, конструкція catch перехопить помилку, та виконає те, що описано всередині. Відбувається логування помилки, щоб зрозуміти причину, чому не відбулась транзакція. В подальшому, програміст зможе виявити причини проблеми та виправити її. Транзакція у разі виникнення помилки не виконується і зміни повертаються у свій попередній стан, до того, як була спроба її виконання.

2.4 Прошарки в сервісній архітектурі

Сервісна архітектура є структурним підходом до розробки програмного забезпечення, в якому додаток розбивається на набір взаємодіючих і незалежних компонентів, названих сервісами. Кожен сервіс представляє собою самостійний функціональний модуль, який виконує конкретні функції та має своє власне API для взаємодії з іншими сервісами. Прошарки в сервісній архітектурі можуть визначати різні аспекти додатку та взаємодії між його компонентами. В кожній команді, розробники самі вирішують, яким чином вони будуть організовувати розбиття проєкту. В залежності від потреб та зручності в масштабуванні проєкту, його розділяють на певні модулі, або ж як раніше було сказано – прошарки. Кожен із таких має на меті організовувати та керувати певними аспектами функціональності системи, роблячи її більш модульною.

Як приклад, прошарки можна розділити наступним чином:

– Прошарок інтерфейсів (API). Визначає API та контракти взаємодії між різними сервісами. Цей прошарок визначає, як інші компоненти можуть спілкуватися з кожним сервісом, надаючи стабільний та чіткий інтерфейс для звертання до функцій. В ньому частіше за все знаходяться контролери, які використовуються для спілкування з користувачем та подальшого виклику необхідних методів сервісу через інтерфейс.
– Прошарок бізнес-логіки. Реалізує бізнес-правила, логіку операцій та обробку даних. Цей прошарок визначає функціональні можливості сервісу та забезпечує їхню правильність та ефективність.
– Прошарок доступу до даних. Забезпечує взаємодію сервісу зі збереженими даними. Це може включати в себе взаємодію з базами даних, кешуванням, а також зовнішніми службами для отримання та оновлення даних.

– Прошарок управління конфігурацією. Управляє конфігурацією сервісів, включаючи налаштування, параметри та інші настройки. Це дозволяє гнучко налаштовувати та масштабувати сервіси без потреби перекомпіляції чи перезапуску.

Вище описані прошарки є загальними, та можуть бути описані по різному. В деяких є можливість реалізації одразу багатьох властивостей прошарків, тобто, забезпечення доступу до даних та конфігурація налаштувань може знаходитись в одному місці.

В проєкті буде використовуватись розподіл на чотири окремих прошарка.

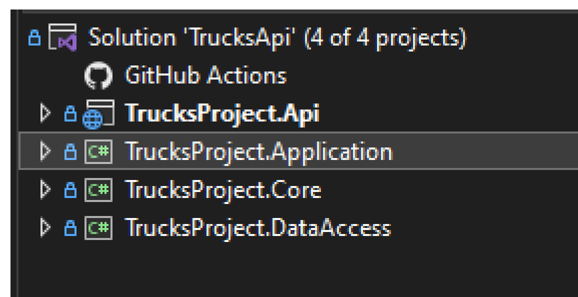


Рис. 2.3 Архітектура проєктного рішення

Перший прошарок – TrucksProject.Api являє собою місцерозташування контролерів, налаштувань проєкту та конвертаторів, які використовуються для контролювання правильності надійдених даних.

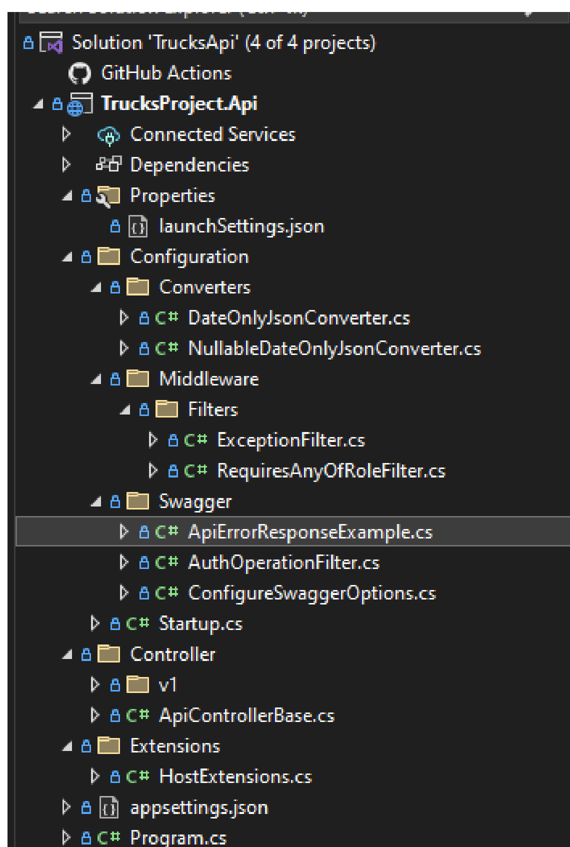


Рис. 2.4 Прошарок “ TrucksProject.Api ”

Кожен клас, який відповідає за певну функціональність, рознесений по відповідним папкам. Папки мають назву відповідно до того, що з себе представляють класи та наповнення якого функціоналу вони мають. Як приклад, в папці Configuration у нас знаходяться класи, які відповідають за налаштування JSON файлу. Він використовується при визначенні налаштувань серіалізації та десеріалізації в механізмі JsonSerializer для роботи з об'єктами типу DateOnly у форматі JSON.

Папка Middleware має в собі класи, які представляють собою фільтри винятків для обробки винятків, які можуть виникнути в процесі обробки HTTP-запитів у додатку. Фільтр спеціально призначений для обробки різних видів винятків і повернення адекватних HTTP-відповідей зі статус-кодами та відповідними повідомленнями. Загалом, вони допомагають забезпечити коректну та стандартизовану обробку винятків у веб-застосунку.

В папці Swagger знаходяться приклади відповідей API для використання в документації Swagger. Він використовує ці приклади для ілюстрації різних сценаріїв відповідей, які можуть бути повернуті під час виклику API.

Папка Controller містить в собі клас ApiControllerBase, який має базові налаштування для контролера. Всі контролери, які будуть в подальшому створюватись, мають обов'язково наслідуватись від цього класу. Таким чином, вирішується проблема, коли один і той самий код пишеться по багато разів. Всі необхідні методи та налаштування вже присутні в цьому класі, тому не буде потреби повторювати їх в нових класах. Тим паче, якщо потрібна буде якась специфічна реалізація певного методу, або зміна налаштувань, програміст зможе змінити реалізацію, не зачепивши при цьому базовий. Вся структура базового класу залишиться такою, яка була описана. Якщо потрібно буде зробити зміни у всіх класах, додавши, наприклад, метод, який точно буде використовуватись в них, такий спосіб знову допоможе нам з цим, адже змінивши тільки базовий, зміни запровадяться у всіх класах-наслідувачах.

В папці Extensions міститься клас, який має конфігурацію для налаштування логування за допомогою бібліотеки Serilog.

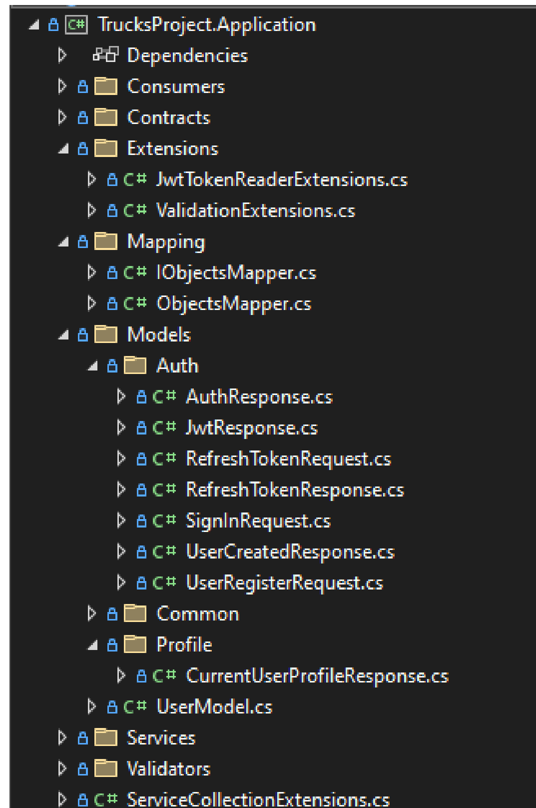


Рис. 2.5 Прошарок “ TrucksProject.Application ”

Цей прошарок відповідає за контракти(інтерфейси), за якими контролери будуть звертатись до методів із сервісу та має опис моделей, за якими буде відбуватися обмін даними між користувачем та сервісом. Окрім цього, тут розташовані і самі сервіси з їх внутрішньою логікою.

Папка Consumers містить клас SampleCommandConsumers, який є споживачем (consumer) повідомлень для шаблону реалізації масштабованих послуг (scalable services) з використанням бібліотеки MassTransit.

Папка Contracts складається з контрактів(інтерфейсів), за якими контролери мають змогу використовувати реалізацію сервісів. Контракти створені в першу чергу для того, аби прошарок з контролерами мав змогу викликати методи, при цьому не знаючи про їхню реалізацію. Таким чином, було зменшено залежність між цими компонентами, а також забезпечимо захищеність від можливих змін. Завдяки такому підходу, контролеру не важливо знати, яким саме чином реалізований той, чи інший метод. Йому

важливий результат, який той має повернути, при цьому подавши йому ті дані, які він вимагає. Зі свого боку, контракт забезпечує повернення результату.

Папка Extensions містить класи, які розширюють функціонал валідатора. Це потрібно для забезпечення перевірки надісланих даних та у разі їх невідповідності, повернути помилку. Таким чином, створюються тільки ті методи, які необхідні в даному прошарку. В інших місцях ці методи використовуватись не будуть, тому вони не є узагальненими, що знову ж таки, дає змогу зменшити зв’язок між прошарками.

Папка Mapping містить правила для мапінгу об’єктів, тобто, це дані або об'єкти одного типу які можуть бути перетворені або "відображені" на об'єкти іншого типу. Це особливо важливо в контексті роботи з об'єктно-реляційними відображеннями (ORM), де виникає потреба в зіставленні даних між різними структурами чи об'єктами.

В папці Models знаходяться моделі, які описують те, які відповіді мають надходити від сервісу. Тобто, формат відповіді, який потім відсилається користувачу, має бути правильно сформований і відображати дані, зрозумілою для користувача мовою. Дані, які отримуються з бази або користувача таким чином можна передавати між собою без створення помилок.

Папка Common містить налаштування для відповіді на сторінці. Розмір сторінки, найменування, зміст самої сторінки для коректного відображення на стороні клієнта.

Папка Profile містить класи з профілями. В даному випадку, у нас присутній профіль Користувача, який має в собі ті ж дані, як і в базі даних і використовується для надання відповіді клієнтові.

В папці Services містяться безпосередньо самі сервіси. Вся їхня логіка, як саме відбуваються операції, пов’язані зі створенням користувача, отриманням даних, змінами чи перевіркою даних. Кожен клас відповідає за певний напрямок, яким він оперує. Наприклад, в папці присутній клас AuthService, який має логіку аутентифікації та авторизації користувача. Окрім цього, в

ньому присутні методи для оновлення токену, або ж для його створення щоб в подальшому повернути користувачу.

В останній папці Validators описані правила для валідації моделей. В даному випадку там присутні валідатори для реєстрації користувача, створення, реєстрації, оновлення токенів, аутентифікації, повернення даних про замовників, перевізників.

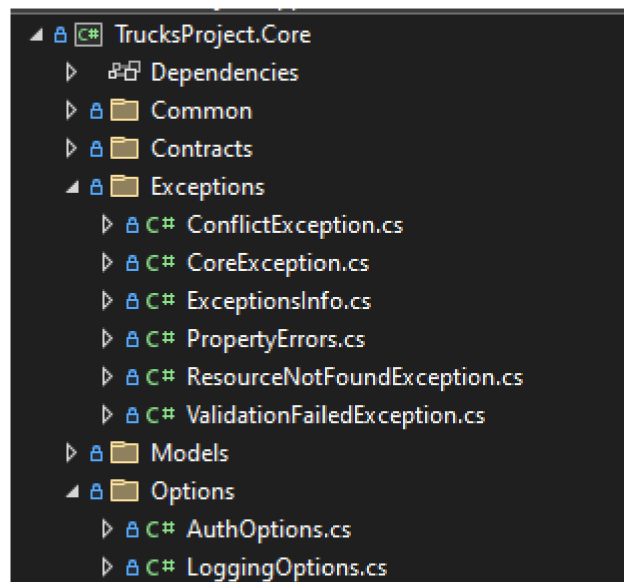


Рис. 2.6 Прошарок “ TrucksProject.Core ”

Цей прошарок відповідає за налаштування того, як будуть описуватись помилки, їх структура, контракти, моделі відповідей та опції аутентифікації та авторизації користувача.

В папці Common знаходяться провайдери, які реалізіують інтерфейсні методи. Вони описують логіку, за якою будуть шифруватися паролі користувача та правильно конвертує час, надісланий з бази або навпаки, для створення єдиного стандарту.

В папці Contracts якраз знаходяться ті інтерфейси, які реалізіуються і папці Common.

Папка Exception має в собі список класів, які описують те, з чого складаються моделі помилок при їх поверненні в контролер. Всі класи, які

мають приписку Excpetion вкінці, наслідуються від класу CoreException, який в свою чергу наслідується від базового Exception.

Папка Models знаходяться моделі, в яких описана структура того, як описується моделі. Які параметри мають бути, змінні для того, аби потім передати це на контролер. Окрім цього, всередині знаходяться ще папки, в яких описуються безпосередньо Entity, які будуть взаємодіяти з базою даних та останнім прошарком - TrucksProject.DataAccess.

Папка Options містить класи, в яких описується структура опцій аутентифікації та авторизації.

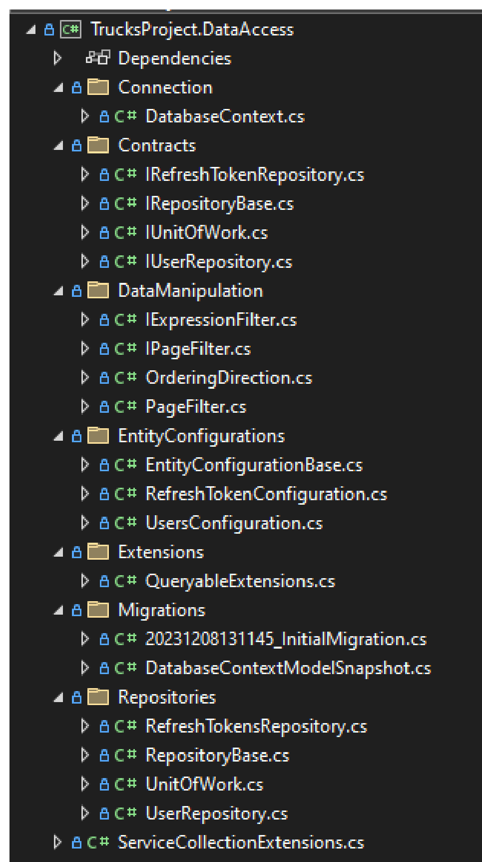


Рис. 2.7 Прошарок “ TrucksProject.Core ”

В останньому прошарку описана вся взаємодія з базою даних. Міграції, конфігурації Entity, репозиторій а також контракти для взаємодії між прошарками.

Папка Connection містить клас DatabaseContext, в якій описані конфігурації для з’єднання з базою даних та налаштування.

Папка Contracts містить контракти, в яких описані методи, які мають в репозиторії. Завдяки цим методам і відбувається записи в базу даних, читання, зміни або видалення. Тут контракти теж слугують для того, аби повідомляти які саме методи мають репозиторії, без показу їхньої реалізації.

Папка DataManipulation містить в собі класи, які маніпулюють даними, які були витягнуті з бази даних. Певні класи можуть правильно відсортувати дані, застосувати фільтри, правильно згрупувати дані, щоб вони вірно відображались на сторінці користувача.

Папка EntityConfigurations має класи з налаштуванням для певних Entity, які присутні в базі.

В папці Extensions міститься клас, який розширює Query – запити, за рахунок того, що динамічно визначає чи слід використовувати відстеження при виконанні LINQ – запитів.

В папці Migrations містяться міграції, які застосовуються для забезпечення консистентності між схемою бази даних та версіями програмного забезпечення. Кожна міграція може представляти конкретний момент часу або версію програми з певними змінами в схемі. Таким чином, є можливість, у разі виникнення помилок або змін архітектури бази даних, повернутися на попередню версію, без втрати даних.

В папці Repositories знаходиться класи, в яких описана логіка взаємодії з базою даних. За допомогою бібліотеки EntityFrameworkCore у нас є змогу використовувати вбудовані методи для взаємодією з базою даних, щоб власноруч не писати кожний запит.

2.5 Розробка методу пошуку найбільш підходящих транспортних засобів серед запропонованих

Для того, аби визначити такі транспортні засоби, нам спочатку потрібно зрозуміти, які саме транспортні засоби для нас – найвигідніші. Розглянемо математичну модель:

$$\sum_{i=1}^{N}\sum_{j=1}^{M} c_{ij} * x_{ij} \rightarrow min \tag{1.1}$$

Згідно неї, визначаються такі транспортні засоби, які зможуть мінімізувати вартість перевезення. До неї також додаються наступні обмеження:

$$\begin{cases} \sum_{j=1}^{M} x_{ij} = 1 \\ \sum_{i=1}^{N}\sum_{j=1}^{M} x_{ij} * w_i \leq cp_j \end{cases} \tag{1.2}$$

де N – кількість вантажів;

M – кількість транспортних засобів;

wi – вага вантажу i;

cpj – вантажопідйомність транспортного засобу j;

cij – вартість перевезення вантажу i транспортним засобом j;

xij – бінарна змінна, яка дорівнює 1, якщо вантаж i перевозиться транспортним засобом j, і 0 в іншому випадку.

Представлені обмеження відповідають за те, що кожен транспортний засіб може перевозити тільки один вантаж, а також, що вага вантажу не має перевищувати вантажопід’ємність самого транспортного засобу. Те, що один транспортний засіб може перевозити лише один товар, має таке обмеження через те, що невідомо що це саме за товар. Можливо, це паливо, яке неможливо

розділити між декількома транспортами, можливо це спеціальний вантаж. Частіше за все, товар перевозять одним перевізником і цілим. Також, вантажівка не може повертатися назад, у неї є лише одне завдання – взяти товар і довезти до пункту прийому. Метод пропонує замовнику оптимальні варіанти серед запропонованих, але це його вибір – погоджуватись з обраними перевізниками, чи ні.

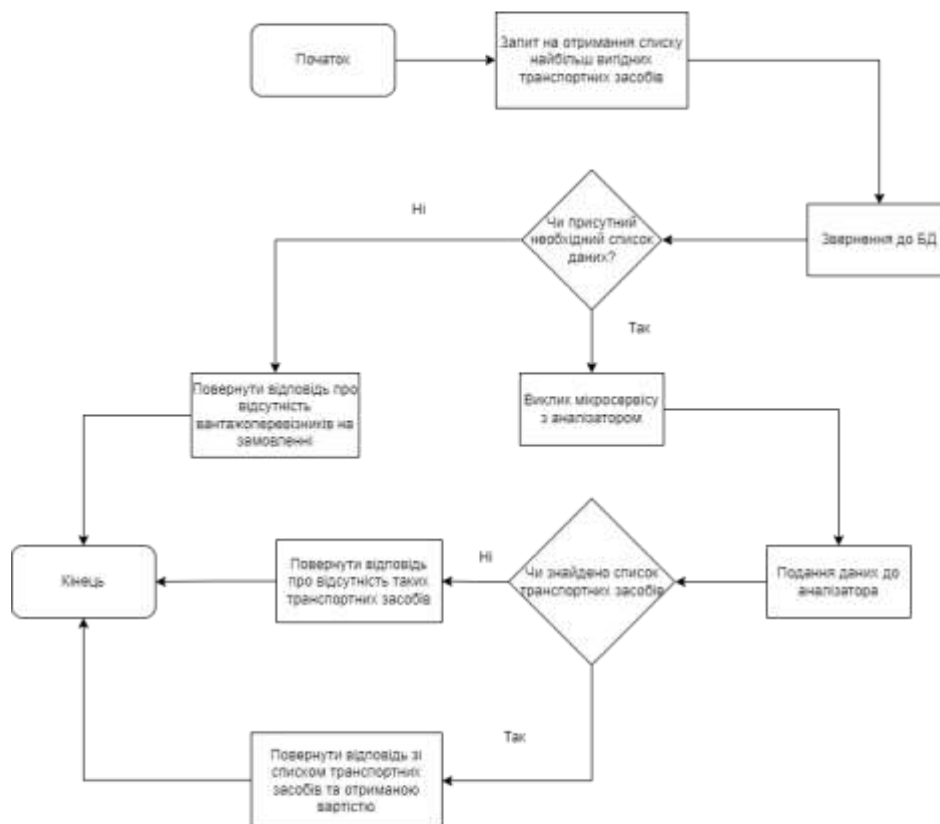


Рис. 2.8 Алгоритм запиту на знаходження оптимальних транспортних засобів

Метод знаходження транспортних засобів буде виконаний на мові програмування Python у вигляді мікросервісу, та викликатися за допомогою відповідного методу в розробленому WEB API на мові програмування C#. На початку, користувач створює запит на отримання списку найбільш вигідних транспортних засобів. Після цього, WEB API отримавши необхідні дані, формує запит в базу даних для перевірки присутності списку перевізників. Якщо список недостатній, сервіс повертає відповідь про відсутність такого списку. Якщо необхідна кількість достатня, йде виклик мікросервісу, в якому

знаходиться метод для знаходження оптимальних транспортних засобів, та подаються необхідні дані. Після відпрацювання самого методу, йде перевірка на те, чи знайдений список транспортних засобів. Якщо такий список знайдено не було, йде повернення відповіді про їх відсутність. В іншому випадку, повертається список транспортних засобів з розрахованою вартістю.

Таким чином, метод допомагає обрати замовнику тих перевізників, які будуть оптимальними для перевезення його вантажу. Також, за допомогою технології WEB API досягнуто того, що вилучили оператора з ланцюжка оформлення замовлення. Тепер замовник сам буде вирішувати якого саме перевізника обрати для його вантажу. Метод знаходження оптимальних транспортних засобів лише пропонує замовнику свої варіанти, і вже безпосередньо замовник має вирішувати, чи погодитись с результатами, або самостійно обрати виходячи з власних потреб та можливостей.

2.6 Порівняння процесу подання заявки замовником на знаходження перевізника через оператора та WEB API

Процес подання замовлення замовником через оператора відрізняється від способу через WEB API. Перше і напевне найголовніша відмінність полягає у тому, що у випадку з оператором, приймає ключову роль та рішення саме він. Яке замовлення оброблювати, якого перевізника обрати, скільки це займатиме часу. Все дуже сильно залежить від людського фактору, присутнього операторові, а також від його досвіду. В деяких випадках, замовлення може бути пропущене, або ж обробка одного замовлення може займати декілька днів. В кінцевому результаті, замовник та перевізник майже ніяк між собою в цьому процесі не комунікують, а повністю покладаються на оператора.

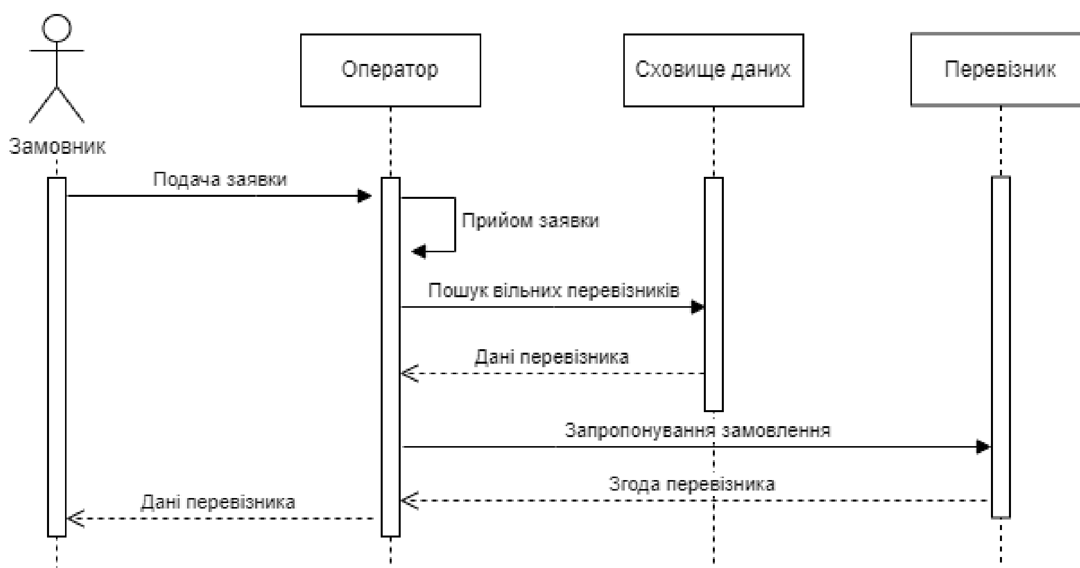


Рис. 2.9 Діаграма послідовностей для подання заявки на перевізника замовником через оператора

В процесі подання заявки через оператора, замовник має зв'язатися з оператором та описати йому всю необхідну інформацію. Оператор в цей час оброблює інформацію, записує всі необхідні дані та після чого виконує запит в сховище даних, де шукає перевізників, які б найбільше підходили на цю роль. Сховище даних в даному випадку може бути різним, починаючи від записів в блокноті закінчуючи базою даних. Від цього також залежить швидкість визначення перевізника. Після отриманих даних, оператор має з'єднатися з перевізником та запропонувати йому замовлення, обговорити всі деталі та у разі згоди передати замовнику дані перевізника.

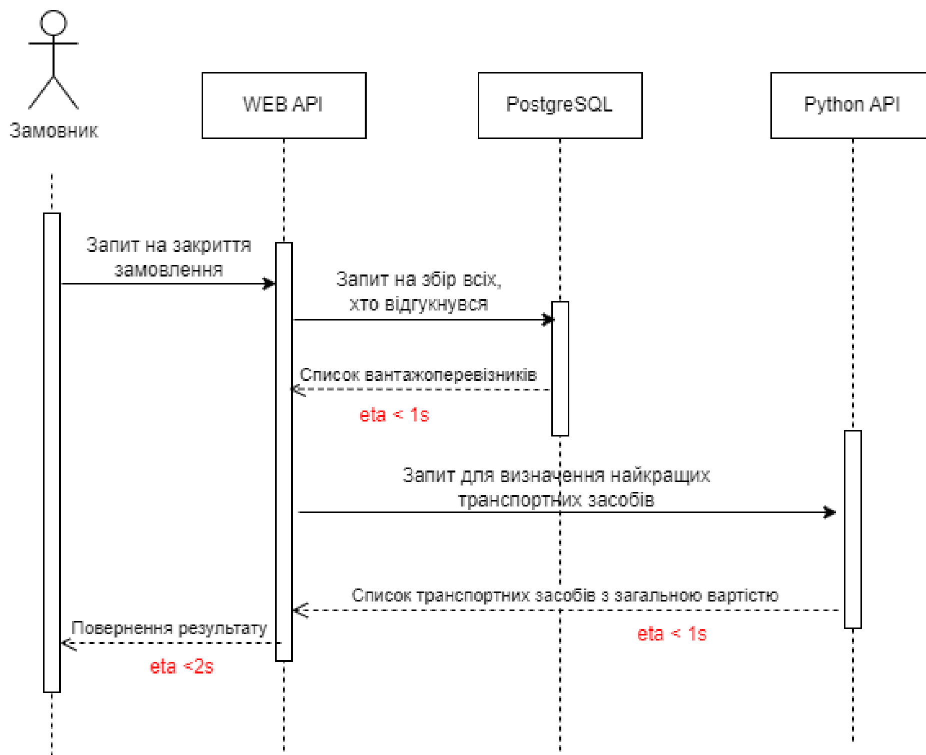


Рис. 2.10 Діаграма послідовностей для визначення замовником найбільш підходящих транспортних засобів

На даній діаграмі зображено взаємодія замовника з WEB API для визначення найбільш підходящих транспортних засобів. Якщо у випадку з оператором вся взаємодія з замовником та перевізником була організована через нього, то у випадку з WEB API що замовник, що перевізник мають можливість переглядати замовлення, у випадку перевізника підписатись на нього, залишивши свої дані.

На діаграмі замовник, який вже створював замовлення, робить запит на його закриття. WEB API виконує запит в базу даних, у цьому випадку це PostgreSQL, на збір всіх тих перевізників, які відгукнулися на дане замовлення. Після отримання списку, викликається мікросервіс, написаний на мові програмування Python, подаючи при цьому необхідні для розрахунків дані. В свою чергу, мікросервіс, отримавши всі необхідні дані, розраховує список оптимальних транспортних засобів та загальну суму перевезення та повертає це на WEB API. З сервісу, відповідь отримує користувач та самостійно вирішує, кого з перевізників наймати для подальшого перевезення вантажу.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ WEB API ДЛЯ ОПТИМІЗАЦІЇ ВЗАЄМОДІЇ МІЖ ЗАМОВНИКОМ ТА ПЕРЕВІЗНИКОМ

3.1 Опис використаних програмних засобів

3.1.1 Visual Studio

Visual Studio представляє собою комплексне інтегроване середовище розробки, розроблене корпорацією Microsoft, яке займає визначне положення серед засобів для створення програмного забезпечення. Його універсальність проявляється в широкому спектрі підтримуваних мов програмування, включаючи, але не обмежуючись, C#, Visual Basic, C++, F#, Python, та інші. Інтегровані інструменти розробки включають редактор коду з різноманітними функціями, такими як підсвічування синтаксису та автоматичне завершення коду, що сприяє ефективному написанню програм.

Однією з ключових характеристик є інтерфейс розробника, який включає графічний дизайнер для створення та редагування елементів графічного інтерфейсу. Це дозволяє розробникам швидко створювати та вдосконалювати користувацький інтерфейс своїх додатків. Крім того, вбудована підтримка систем керування версіями, таких як Git, дозволяє командам розробників ефективно співпрацювати та відстежувати зміни в коді.

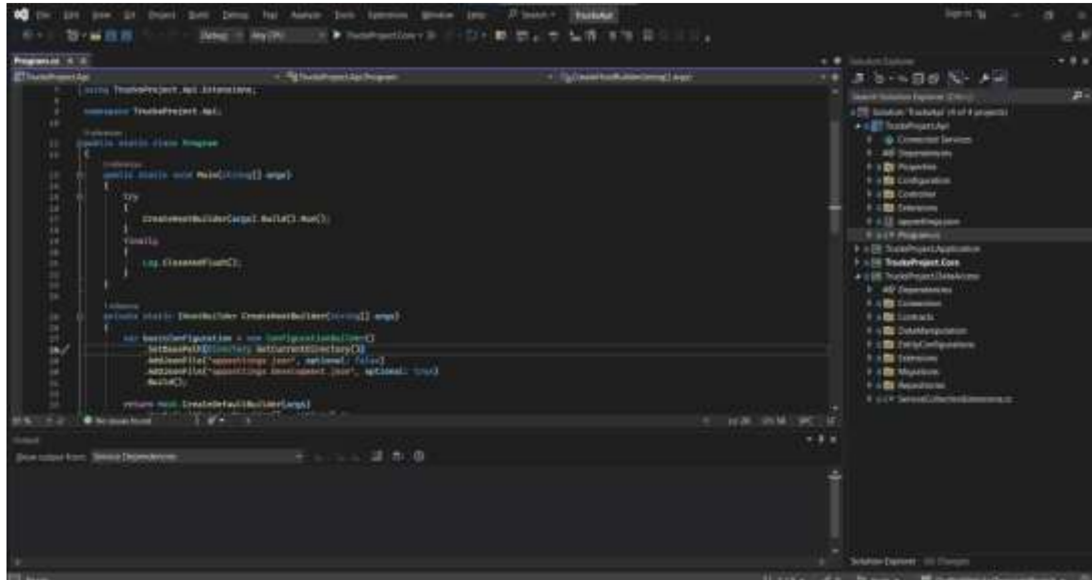


Рис. 3.1 Інтерфейс IDE Visual Studio

Visual Studio не обмежується платформою, але забезпечує розробку для різних середовищ, включаючи Windows, веб-розробку та мобільні додатки (за допомогою Xamarin). Для полегшення роботи в команді в інструментарії Visual Studio також вбудовані інструменти для відлагодження та профілювання, що сприяють виявленню помилок та оптимізації продуктивності коду.

Особливу увагу слід приділити розширюваності Visual Studio, де розробники можуть використовувати власні розширення або користуватися розширеннями від інших розробників для покращення функціональності IDE. Крім того, інтеграція з хмаровими службами, зокрема Azure, дозволяє легко розгортати та управляти хмарними додатками.

Visual Studio є потужним інструментом для розробників, який надає широкий набір можливостей та допомагає створювати високоякісне програмне забезпечення на різних платформах та мовах програмування.

3.1.2 Мова програмування C#

Мова програмування C# (C-Sharp) визначається як одна з ключових складових екосистеми розробки Microsoft та широко використовується в інтегрованому середовищі розробки Visual Studio. Заснована на принципах

об'єктно-орієнтованого програмування, C# визначається як мова, спрямована на зручність розробників та безпеку виконання коду.

Однією з ключових переваг C# є його висока простота та зрозумілість синтаксису, що робить його досить доступним для розробників різних рівнів кваліфікації. Інтеграція C# з Visual Studio забезпечує розширений редактор коду з автоматичним завершенням, відлагодженням та іншими інструментами, спрямованими на полегшення процесу розробки.

Важливим елементом мови є підтримка мультитредінгу та асинхронного програмування, що робить C# потужним інструментом для створення продуктивних та ефективних додатків. Інша суттєва характеристика - автоматичне управління пам'яттю (garbage collection), що полегшує завдання розробників та зменшує ймовірність виникнення помилок управління пам'яттю.

C# також відомий своєю високою портативністю, що означає можливість використання коду на різних платформах, адаптуючи його під конкретні вимоги. Мова взаємодіє з різними технологіями, включаючи технології хмарового обчислення, мобільні платформи та веб-розробку.

Розширена підтримка мови спрощує створення високоякісних графічних інтерфейсів користувача за допомогою технології Windows Presentation Foundation (WPF). Окрім того, C# має активну спільноту розробників та постійно оновлюється, що робить її сучасною та конкурентоздатною мовою програмування.

Варто зазначити, що сама мова програмування постійно розвивається, додаються нові можливості та вдосконалення. Завдяки відкритому коду фреймворку .NET CORE користувачі мають змогу запропонувати власні рішення для того, аби покращити мову програмування. Чим більше залученість аудиторії, тим більш досконалими виходять рішення. Завдяки таким пропозиціям від користувачів, розробники додають їх вже в основну версію, що дає змогу використовувати всі сучасні рішення без попереднього їх пошуку та встановлення. Також, якщо розробники додають таке рішення в основну

версію, отже воно пройшло досить серйозну перевірку та може без проблем бути використаним для впровадження у власні рішення.

3.1.3 Figma

Figma - це інноваційний інструмент для дизайну та прототипування, який завоював популярність серед дизайнерів і тім-лідів завдяки своїй унікальній хмарній архітектурі та відмінним можливостям спільної роботи. Забезпечуючи візуальне проектування інтерфейсів та взаємодію між елементами, Figma став необхідним інструментом для розробників, що працюють у сферах веб-дизайну, UX/UI-дизайну та розробки мобільних додатків.

Однією з ключових особливостей Figma є можливість роботи в реальному часі в хмарному середовищі. Це дозволяє розробникам та дизайнерам працювати над проектом одночасно, без необхідності синхронізації або експорту файлів. Будь-яка зміна, яку внесено в дизайн, відображається миттєво для всіх членів команди, сприяючи ефективній комунікації та спільній роботі.

Інтерфейс Figma дружній та інтуїтивний, забезпечуючи різноманітні інструменти для створення макетів, інтерактивних прототипів та редагування векторної графіки. Сервіс також дозволяє робити зручні зауваження, анотації та коментарі, спрощуючи обговорення проекту в команді.

Figma впроваджує концепцію компонентів, яка дозволяє створювати повторно використовувані елементи та шаблони, що зберігають єдність дизайну на різних сторінках чи екранах. Це сприяє швидкому та узгодженому розвитку проекту.

Підтримка для векторної графіки, растрових зображень, анімацій та інших функцій робить Figma потужним інструментом для повноцінного створення та тестування дизайну перед його втіленням в коді. Також, інтеграція з іншими інструментами розробки та перевірка у реальних умовах допомагають у підтримці синхронізації між дизайном та реалізацією.

Figma став важливим інструментом для розробників та дизайнерів, забезпечуючи швидкий, ефективний та спільний процес проєктування та прототипування в умовах сучасного розвитку програмного забезпечення.

При проєктування додатку та бази даних, Figma використовувалась для проєктування можливих сторінок, які могли бути при використанні розробленого WEB API. Це було зроблено для того, аби розуміти, яким чином розроблювати моделі в сервісі, які значення повертати та які валідатори даних при цьому використовувати. В подальшому стало зрозуміло, що може виникнути проблема з формування JSON – файлу, і шо його теж потрібно було валідувати. Не менш важливим було і збереження дати в системі, адже сайт міг подати просто не той формат, через що могла виникнути помилка, коли дата записувалась невірним числом. WEB API все ще залишається по своїй суті API, але прогнозування та проєктування таких сторінок допомогло проаналізувати ті помилки, які могли виникнути та уникнути їх.

3.1.4 Postman

Postman – це відмінний інструмент, який став необхідним компаньйоном для розробників та тестувальників у всьому світі. Цей інструмент, який призначений для розробки та тестування програмного забезпечення, відзначається своєю виразністю у полегшенні взаємодії з API, і надає безліч корисних можливостей для зручності та ефективності розробки.

Однією з найважливіших переваг Postman є його інтуїтивний та дружелюбний інтерфейс, що дозволяє користувачам створювати та відправляти різні HTTP-запити, такі як GET, POST, PUT, DELETE, з легкістю. Можливість керувати параметрами запиту, включаючи заголовки та тіло запиту, робить взаємодію з API приємною та ефективною.

Постійне оновлення та вдосконалення Postman дозволяє розробникам автоматизовувати тестові сценарії, перевіряти відповіді сервера та забезпечувати стабільність свого програмного забезпечення. Можливість

організовувати запити в колекції полегшує структурування та збереження проектів, особливо в тім-лідів та командних середовищах.

Postman також надає можливості для обміну та спільної роботи, дозволяючи розробникам ділитися своїми колекціями, щоб команда могла спільно працювати над різними аспектами проекту. Можливість зберігання та обміну змінними середовища сприяє швидкій адаптації та повторному використанню даних.

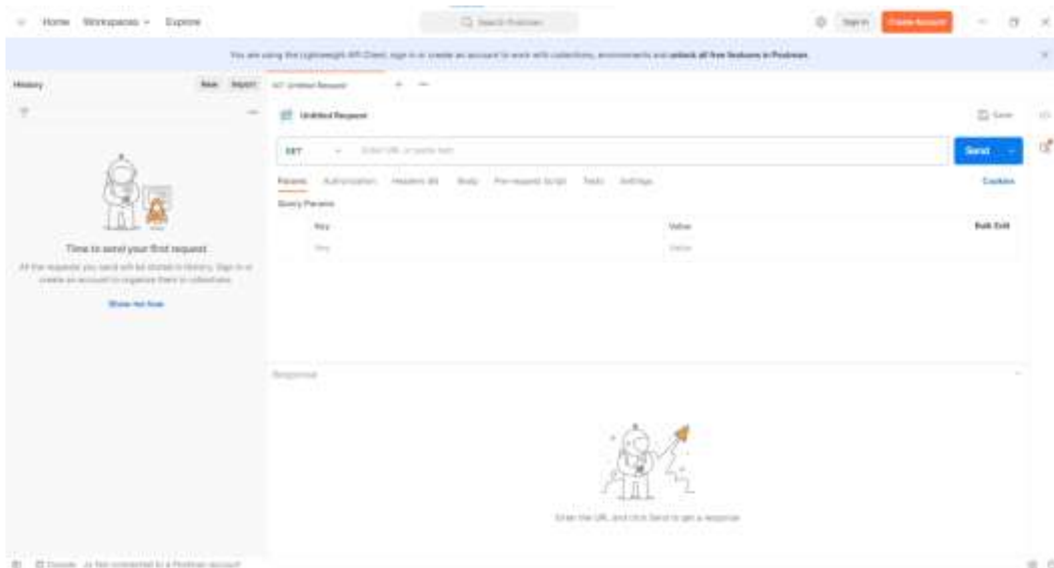


Рис. 3.2 Інтерфейс Postman

Postman також вигідно відзначається функціоналом моніторингу та документації API. Здатність створювати документацію API та візуалізувати виклики дозволяє командам легше спілкуватися та розуміти характеристики API.

Завдяки підтримці різних методів автентифікації, включаючи токени та базову автентифікацію, Postman дозволяє взаємодіяти з захищеними API без зайвих труднощів.

3.1.5 Docker

Docker - це засіб для розробки та розгортання програмного забезпечення. Цей інструмент визначається не лише технічними можливостями, але й

впливом на весь життєвий цикл розробки, забезпечуючи гнучкість, ефективність та зручність у кожному кроці.

Однією з ключових переваг Docker є його надзвичайна зручність у використанні. Створення та управління контейнерами стає простим завдяки інтуїтивному інтерфейсу та можливості швидко переносити програми та їхні залежності від одного середовища до іншого. Це є особливо цінним в світі розробки, де різні команди працюють на різних операційних системах.

Контейнеризація в Docker дозволяє розробникам та тестувальникам створювати стандартизовані, відокремлені середовища, забезпечуючи консистентність у різних етапах розробки. Це також робить процес розгортання більш простим, оскільки всі необхідні компоненти вже включені в контейнер.

Рис. 3.3 Інтерфейс Docker

Найцікавіше те, що Docker дозволяє ефективно використовувати ресурси, забезпечуючи ізоляцію контейнерів та можливість їхнього швидкого запуску. Це дозволяє розробникам швидко переходити від розробки до тестування та розгортання, зберігаючи при цьому стабільність додатку.

Інтеграція Docker з різними інструментами розробки, такими як системи керування версіями та інструменти для неперервної інтеграції та доставки, додає ще один шар зручності в розробничий процес. А взаємодія з

оркестраторами контейнерів, наприклад Kubernetes, взагалі перетворює Docker у незамінного гравця у розвитку масштабних та розподілених систем.

3.1.6 GitHub

GitHub — це масштабна та універсальна платформа, що визначає нові стандарти у сфері розробки програмного забезпечення та співпраці розробників. Здобувши шалену популярність, вона стала не просто інструментом, але насправді невід'ємною частиною життєвого циклу будь-якого проєкту.

Найбільша цінність GitHub, безумовно, полягає в системі керування версіями Git, що дозволяє ефективно відстежувати зміни у коді. Репозиторії на GitHub служать не лише сховищем для вихідного коду, але і центральним місцем для обговорень, взаємодії та співпраці між розробниками.

Один з ключових аспектів GitHub - галузь розгалужень (гілок). Розробники можуть створювати окремі гілки для роботи над конкретними функціональностями чи розв'язанням проблем. Це не лише сприяє паралельній розробці, але й надає зручний механізм для об'єднання внесених змін за допомогою запитів на злиття.

Запити на злиття — це інша потужна можливість GitHub, яка дозволяє розробникам пропонувати та обговорювати зміни перед їхнім включенням у основний код проекту. Це ефективний механізм код-рев'ю, який допомагає підтримувати високу якість коду та виявляти можливі проблеми.

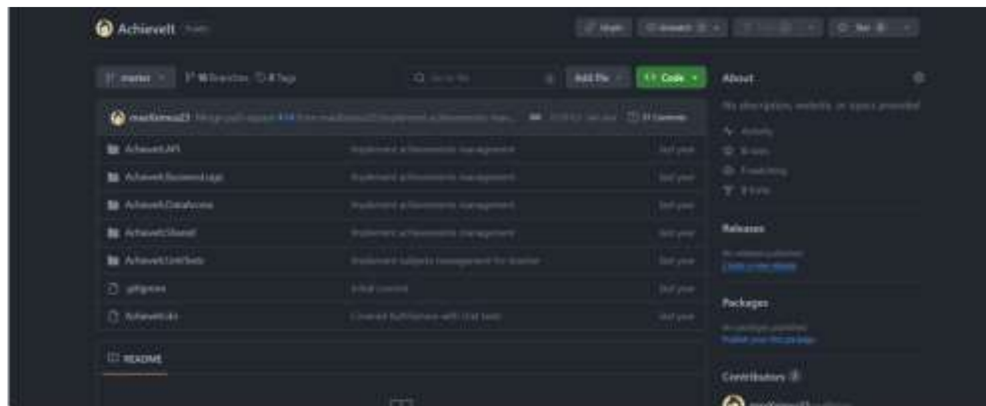


Рис. 3.4 Інтерфейс сайту GitHub

Зручність ведення списків проблем та завдань, а також їхнє призначення розробникам, робить GitHub ефективним інструментом для управління проектами та взаємодії команд. Автоматизовані процеси, включаючи неперервну інтеграцію та розгортання (CI/CD) завдяки інтеграції з різноманітними сервісами, роблять розробку швидшою та надійнішою.

GitHub Actions додають новий рівень автоматизації, дозволяючи визначати та спільно використовувати власні робочі процеси прямо в репозиторії. Це може охоплювати всі аспекти від тестування до розгортання, роблячи розробку ще більш ефективною.

Однак GitHub - це не просто технічний інструмент. Він також став важливим соціальним майданчиком для розробників. Можливість стежити за іншими розробниками, взаємодіяти через коментарі та виражати визнання за допомогою зірок роблять GitHub спільнотою, яка допомагає розробникам взаємодіяти, вчитися та спільно розвивати нові технології.

3.1.7 PyCharm

PyCharm - це інтегроване середовище розробки (IDE) для програмування на мові Python, створене компанією JetBrains. Воно забезпечує розширені інструменти для підтримки розробки, включаючи рефакторинг, аналіз коду, автоматичне завершення коду та інші корисні функції.

Однією з ключових особливостей PyCharm є висока інтеграція з іншими інструментами та технологіями, яка полегшує роботу розробників. У нього є потужний редактор коду, засоби аналізу коду, підтримка віртуальних середовищ, зручний інтерфейс відладки, можливість керування залежностями та підтримка веб-технологій.

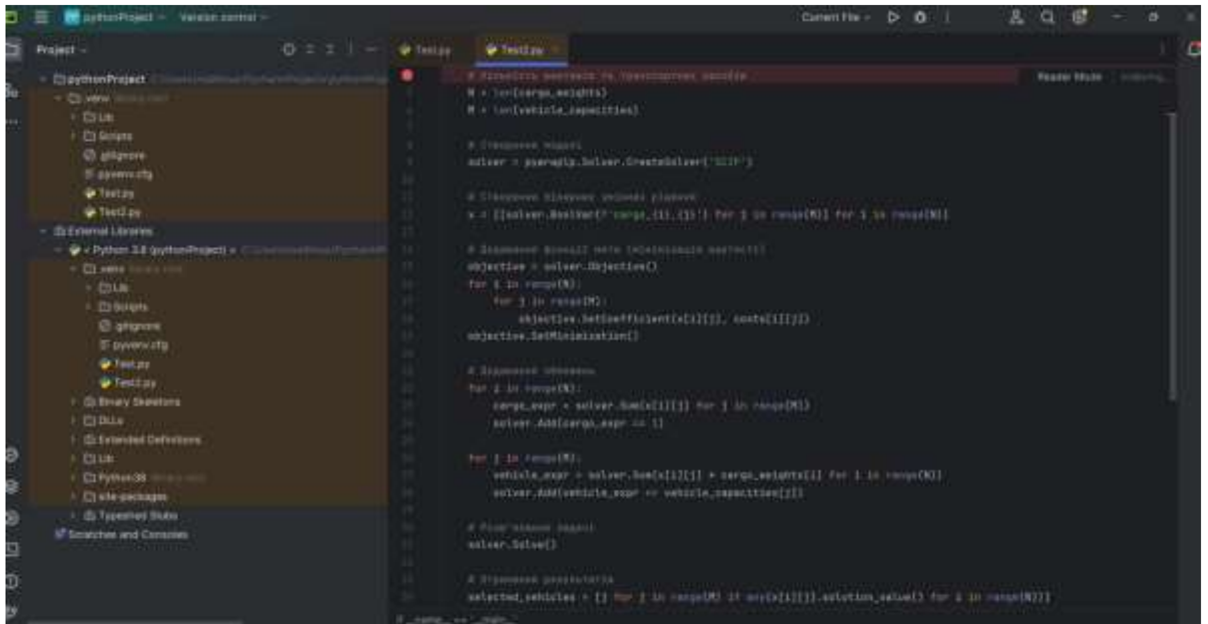


Рис. 3.5 Інтерфейс IDE PyCharm

PyCharm також має інтеграцію з системами керування версіями, зокрема Git, а також з фреймворками, такими як Django. Він надає інструменти тестування, спрощуючи написання та виконання тестів. Для розробників, які працюють з веб-додатками, PyCharm пропонує підтримку мов HTML, CSS та JavaScript.

Інші важливі риси PyCharm включають можливість розширення за допомогою плагінів та інструменти для роботи з фреймворками, такими як Django. Усе це робить PyCharm потужним інструментом для ефективної розробки на мові Python, забезпечуючи розробникам необхідні засоби для написання, відлагодження та управління кодом.

3.1.8 Мова програмування Python

Однією з ключових особливостей Python є його чистий та зрозумілий синтаксис, що дозволяє розробникам виражати ідеї в менше кількості рядків коду порівняно з іншими мовами. Мова використовує відступи для визначення блоків коду, що робить код більш структурованим і чітким.

Python підтримує об'єктно-орієнтоване програмування, функціональне програмування, аспекто-орієнтоване програмування та інші парадигми. Велика

кількість вбудованих бібліотек та модулів робить його потужним інструментом для різних задач.

Мова є переносною, що означає, що програми, написані на Python, можуть використовуватися на різних операційних системах без змін. Це робить його популярним в області розробки крос-платформенних додатків.

Python має велике та активне спільнотою розробників, яка підтримує розвиток та вдосконалення мови. Інтерпретований характер мови дозволяє виконувати код безпосередньо, що спрощує процес розробки та експериментів.

Важливою рисою Python є його розширюваність. Розробники можуть використовувати сторонні бібліотеки та модулі для розширення можливостей мови. Це включає в себе багато галузей, таких як наука про дані, штучний інтелект, веб-розробка та інше.

Узагальнюючи, Python є універсальною, динамічною та ефективною мовою програмування, яка знаходить широке застосування у різних сферах розробки програмного забезпечення.

3.2 Опис класів

В системі присутні багато класів, які відповідають за свій певний функціонал. В залежності від того, де знаходиться сам клас(в якому прошарку та папці), він має відповідати певним вимогам та наповненню функціоналу. Наприклад, якщо клас знаходиться в прошарку з контролерами, не слід в ньому описувати взаємодію з базою даних, або намагатися виконати те, для чого цей прошарок не призначений.

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Serilog;
using TrucksProject.Api.Configuration;
using TrucksProject.Api.Extensions;

namespace TrucksProject.Api;

0 references
public static class Program
{
    0 references
    public static void Main(string[] args)
    {
        try
        {
            CreateHostBuilder(args).Build().Run();
        }
        finally
        {
            Log.CloseAndFlush();
        }
    }

    1 reference
    private static IHostBuilder CreateHostBuilder(string[] args)
    {
        var basicConfiguration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json", optional: false)
            .AddJsonFile("appsettings.Development.json", optional: true)
            .Build();

        return Host.CreateDefaultBuilder(args)
            .UseDefaultServiceProvider((_, options) =>
            {
                options.ValidateScopes = true;
                options.ValidateOnBuild = true;
            })
            .AddLogging(basicConfiguration)
```

Рис. 3.6 Клас Program.cs

Клас Program.cs слугує як точка входу для запуску ASP.NET Core додатка, і він ініціалізує необхідні складові для коректного функціонування додатка. Метод Main() викликає CreateHostBuilder() для створення та налаштування IHostBuilder, а потім викликає Build() для побудови IHost та Run() для запуску додатка. У блоках try-finally закривається інстанція Log, щоб завершити логування перед завершенням програми. CreateHostBuilder метод налаштовує сервіс-контейнер та додає налаштоване логування з конфігурацією. Також він конфігурує веб-хостбілдер та вказує клас Startup для налаштування веб-додатка.

```
3 references
public class Startup
{
    private const string MainCorsPolicy = "MainPolicy";

    private readonly IConfiguration _configuration;
    private readonly IWebHostEnvironment _environment;

    0 references
    public Startup(IConfiguration configuration, IWebHostEnvironment environment)
    {
        _configuration = configuration;
        _environment = environment;
    }

    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddHttpContextAccessor();

        services.AddFriendlyJwt();
        services.Configure<AuthOptions>(_configuration.GetSection(nameof(AuthOptions)));

        services.AddDataAccessServices(_configuration, _environment);
        services.AddApplicationServices(_configuration);
        services.AddBroker();

        services.AddRouting(options => options.LowercaseUrls = true);

        services.AddApiVersioning(options =>
        {
            options.DefaultApiVersion = new Microsoft.AspNetCore.Mvc.ApiVersion(1, 0);
            options.AssumeDefaultVersionWhenUnspecified = true;
            options.ReportApiVersions = true;

            options.ApiVersionReader = ApiVersionReader.Combine(
                new UrlSegmentApiVersionReader(),
                new HeaderApiVersionReader("x-api-version"),
                new MediaTypeApiVersionReader("x-api-version")
```

Рис. 3.7 Клас Startup.cs

Цей клас Startup визначає конфігурацію та послуги для ASP.NET Core додатка.

Нижче наведено опис його функцій:

– Конструктор. Приймає IConfiguration та IWebHostEnvironment для доступу до конфігурації та середовища відповідно.
– ConfigureServices: Налаштовує та реєструє різноманітні служби.
– Configure. Налаштовує опції додатка та використовує маршрутизацію. Включає налаштування CORS, аутентифікацію та авторизацію. Додає Swagger для генерації документації API та встановлює кілька ендпоінтів.

Сам метод ConfigureServices відповідає за налаштування наступного функціоналу:

– Доступ до HTTP контексту.
– Валідація JWT-токенів.
– Сервіси доступу до даних.

- Додаткові конфігураційні та додаткові сервіси за допомогою методів розширення.
- Користувацькі конвертери JSON для об'єктів, які не є стандартними для серіалізації JSON.
- Аутентифікація за допомогою JWT та обробка винятків через фільтри.
- Додавання Swagger та налаштування його опцій, включаючи версіонування API та приклади для документації.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using TrucksProject.Core.Exceptions;
using TrucksProject.Core.Models.Api;

namespace TrucksProject.Api.Configuration.Middleware.Filters;

internal sealed class ExceptionFilter : ExceptionFilterAttribute
{
    private readonly ILogger<ExceptionFilter> _logger;
    private readonly IHostEnvironment _environment;

    public ExceptionFilter(ILogger<ExceptionFilter> logger, IHostEnvironment environment)
    {
        _logger = logger;
        _environment = environment;
    }

    public override void OnException(ExceptionContext context)
    {
        HandleExceptionAsync(context);
        context.ExceptionHandled = true;
    }

    private void HandleExceptionAsync(ExceptionContext context)
    {
        switch (context.Exception)
        {
            case ResourceNotFoundException:
                SetExceptionResult(context, StatusCodes.Status404NotFound);
                break;
            case ValidationFailedException:
                SetExceptionResult(context, StatusCodes.Status400BadRequest);
                break;
```

Рис. 3.8 Клас ExceptionFilter.cs

Клас ExceptionFilter є фільтром винятків для обробки та відповідного відображення помилок в API. Основні функції включають обробку різних типів винятків, генерацію відповідей з помилковими даними та налаштування цього процесу.

```
1 reference
11  public sealed class RequiresAnyOfRoleFilter : Attribute, IAuthorizationFilter
12  {
        2 references
13      private string[] AllowedRoles { get; init; }
14
        0 references
15      public RequiresAnyOfRoleFilter(params Role[] allowedRoles)
16      {
17          AllowedRoles = allowedRoles.Select(r => r.ToString().ToLowerInvariant()).ToArray();
18      }
19
        0 references
20      public void OnAuthorization(AuthorizationFilterContext context)
21      {
22          var jwtTokenReader = context.HttpContext.RequestServices.GetRequiredService<IJwtTokenReader>();
23          if (!jwtTokenReader.IsLoggedIn)
24          {
25              context.Result = new UnauthorizedResult();
26          }
27
28          var roles = jwtTokenReader.UserRoles;
29          var hasRole = roles.Any(r => AllowedRoles.Contains(r.ToLowerInvariant()));
30          if (!hasRole)
31          {
32              context.Result = new ForbidResult();
33          }
34      }
35  }
```

Рис. 3.9 Клас RequiresAnyOfRoleFilter.cs

RequiresAnyOfRoleFilter є класом, що використовується для реалізації фільтра авторизації в ASP.NET Core. Його призначення полягає в контролі доступу до ресурсів на основі ролей користувачів, що визначаються у токені.

Клас включає в себе приватне поле AllowedRoles, яке зберігає перелік дозволених ролей. Цей перелік ініціюється через конструктор, який приймає параметр allowedRoles. Ролі перетворюються в рядки та зберігаються в нижньому регістрі для подальшого порівняння.

Основний метод класу - OnAuthorization, реалізується з інтерфейсу IAuthorizationFilter. У цьому методі відбувається перевірка наявності дійсного токену. Якщо користувач не автентифікований (тобто відсутній дійсний токен), встановлюється результат UnauthorizedResult, що вказує на відмову в доступі.

Після цього проводиться перевірка, чи має користувач хоча б одну з дозволених ролей. Якщо ні, встановлюється результат ForbidResult, що вказує на відмову у доступі через недостатність необхідних ролей.

Загальна ідея цього класу - це забезпечити механізм контролю доступу на рівні ролей користувачів, використовуючи JWT токени. Це дозволяє гнучко конфігурувати права доступу для різних частин додатка, обмежуючи або відкриваючи доступ до конкретних функцій чи ресурсів в залежності від ролей користувачів.

```csharp
using System.Collections.Generic;
using Swashbuckle.AspNetCore.Filters;
using TrucksProject.Core.Exceptions;
using TrucksProject.Core.Models.Api;

namespace TrucksProject.Api.Configuration.Swagger;

public sealed class ApiErrorResponseExample : IMultipleExamplesProvider<ApiErrorResponse>
{
    public IEnumerable<SwaggerExample<ApiErrorResponse>> GetExamples()
    {
        yield return SwaggerExample.Create(
            "Internal validation failed",
            new ApiErrorResponse(ExceptionsInfo.Identifiers.ValidationFailed, new[]
            {
                new ApiErrorResponseNode(null, "Here the error message.")
            }));

        yield return SwaggerExample.Create(
            "Request model validation failed",
            new ApiErrorResponse(ExceptionsInfo.Identifiers.ModelValidationFailed, new[]
            {
                new ApiErrorResponseNode("PropertyName", new[] { "Here the error message.", "And another error message." }),
                new ApiErrorResponseNode("OtherPropertyName", "Here the error message."),
            }));

        yield return SwaggerExample.Create(
            "Resource not found",
```

Рис. 3.10 Клас ApiErrorResponseExample.cs

ApiErrorResponseExample є класом, призначеним для надання прикладів відповідей на помилки для використання в документації Swagger (OpenAPI) в ASP.NET Core.

Цей клас реалізує інтерфейс IMultipleExamplesProvider<ApiErrorResponse>, що вказує на те, що він відповідає за надання кількох прикладів для класу ApiErrorResponse. Клас ApiErrorResponse використовується для представлення відповідей на помилки в API.

Метод GetExamples визначає різні приклади відповідей на помилки, які можуть виникнути під час використання API. Кожен приклад представляється як об'єкт SwaggerExample<ApiErrorResponse>, що містить опис сценарію та відповідний об'єкт ApiErrorResponse.

Наприклад, один з прикладів вказує на те, що виникла внутрішня помилка валідації, і включає об'єкт ApiErrorResponse з ідентифікатором помилки та списком ApiErrorResponseNode, який представляє вузли відповіді на помилку.

Інші приклади включають сценарії, де виникла помилка валідації запитованої моделі, ресурс не знайдено або сталася неочікувана помилка.

Загалом, клас ApiErrorResponseExample використовується для генерації прикладів відповідей на помилки, які можна використовувати в документації

Swagger для детального пояснення можливих сценаріїв помилок, що можуть виникнути під час використання API.

```
namespace TrucksProject.Api.Configuration.Swagger;

public sealed class ConfigureSwaggerOptions : IConfigureNamedOptions<SwaggerGenOptions>
{
    private readonly IApiVersionDescriptionProvider _provider;

    public ConfigureSwaggerOptions(IApiVersionDescriptionProvider provider)
    {
        _provider = provider;
    }

    /// <summary>
    /// Configure each API discovered for Swagger Documentation.
    /// </summary>
    public void Configure(SwaggerGenOptions options)
    {
        // Add swagger document for every API version discovered
        foreach (var description in _provider.ApiVersionDescriptions)
        {
            options.SwaggerDoc(description.GroupName, CreateVersionInfo(description));
        }
    }

    /// <summary>
```

Рис. 3.11 Клас ConfigureSwaggerOptions.cs

ConfigureSwaggerOptions є класом, відповідальним за налаштування опцій для Swagger (OpenAPI) в ASP.NET Core додатках.

Клас реалізує інтерфейс IConfigureNamedOptions<SwaggerGenOptions>, що вказує на його функціональність налаштування для об'єкта SwaggerGenOptions. В конструкторі він отримує інтерфейс IApiVersionDescriptionProvider, який використовується для отримання інформації про доступні версії API.

Метод Configure викликається для налаштування опцій Swagger. У цьому методі для кожної версії API, яка визначена IApiVersionDescriptionProvider, додається документ Swagger (OpenApiDocument). Кожен документ Swagger отримує назву, що відповідає групі версій API, та інформацію про версію API, яку створює метод CreateVersionInfo.

Метод Configure також реалізує перевантажений варіант з іменем, який викликає основний метод Configure.

Метод CreateVersionInfo створює об'єкт OpenApiInfo, який містить інформацію про версію API. Задається назва API ("Trucks API") та номер версії.

Якщо версія є застарілою (deprecated), до опису інформації про версію додається відповідне повідомлення.

Цей клас додає гнучкість та автоматизацію при роботі з Swagger в ASP.NET Core, забезпечуючи створення документації для кожної версії API, а також додаючи інформацію про версії та їх стан (наприклад, застарілість).

```
namespace TrucksProject.Core.Models.Api;

public class ApiErrorResponse
{
    public ApiErrorResponse(string errorType, ApiErrorResponseNode[] errorNodes)
    {
        ErrorType = errorType;
        Errors = errorNodes;
    }

    public ApiErrorResponse(string errorType, ApiErrorResponseNode errorNode)
    {
        ErrorType = errorType;
        Errors = new[] { errorNode };
    }

    public string ErrorType { get; init; }
    public ApiErrorResponseNode[] Errors { get; init; }
}
```

Рис. 3.12 Клас ApiErrorResponse.cs

ApiErrorResponse представляє собою клас для моделювання відповіді на помилку API. Цей клас призначений для створення структурованої відповіді, що містить інформацію про тип помилки та пов'язані з нею деталі.

Клас має два конструктори, які приймають різні комбінації параметрів для створення екземпляру класу. Перший конструктор отримує тип помилки (errorType) та масив об'єктів ApiErrorResponseNode, що представляє вузли з деталями про помилку. Другий конструктор приймає також тип помилки, але в даному випадку передає лише один об'єкт ApiErrorResponseNode, щоб спростити створення екземпляру для одиночних помилок.

Поля ErrorType та Errors визначаються як властивості з модифікатором init, що дозволяє їх ініціалізувати тільки під час створення об'єкта чи виразу ініціалізації. Це важливо для забезпечення незмінності об'єкта після його створення.

Клас ApiErrorResponse призначений для використання у відповідях API, де може виникнути необхідність передавати докладну інформацію про помилку, що сталася під час виклику API. Він спрощує процес формування та передавання даних про помилку, що сприяє читабельності та розумінню відповідей API для розробників та інших користувачів.

```
public sealed class UnitOfWork : IUnitOfWork, IDisposable
{
    private readonly DatabaseContext _databaseContext;
    private readonly ILogger<UnitOfWork> _logger;
    private readonly IServiceScope _serviceScope;

    private IUserRepository _userRepository;
    private IRefreshTokenRepository _refreshTokenRepository;

    public UnitOfWork(IServiceProvider serviceProvider)
    {
        _serviceScope = serviceProvider.CreateScope();
        _logger = GetService<ILogger<UnitOfWork>>();
        _databaseContext = GetService<DatabaseContext>();
    }

    public IUserRepository Users => _userRepository ??= GetService<IUserRepository>();
    public IRefreshTokenRepository RefreshTokens => _refreshTokenRepository ??= GetService<IRefreshTokenRepository>();

    public async Task CommitTransactionAsync(Action action)
    {
        await using var transaction = await _databaseContext.Database.BeginTransactionAsync();
```

Рис. 3.13 Клас UnitOfWork.cs

UnitOfWork є класом, який реалізує інтерфейс IUnitOfWork та відповідає за керування транзакціями та взаємодію з репозиторіями даних у контексті роботи з базою даних. Давайте розглянемо його функціональність та основні характеристики.

Клас має приватні поля, які використовуються для зберігання екземплярів DatabaseContext (контексту бази даних), ILogger<UnitOfWork> (логгера) та IServiceScope (області сервісів), яка використовується для створення нового області сервісів. Також містить два репозиторії для роботи з об'єктами користувачів та токенами оновлення.

Конструктор класу отримує IServiceProvider, який використовується для створення області сервісів та отримання необхідних сервісів.

Метод CommitTransactionAsync приймає делегат Action або Func<Task>, який виконується в межах транзакції. Цей метод дозволяє виконати дії, які повинні бути атомарними, тобто вони виконуються повністю або не

виконуються зовсім. Метод також забезпечує логування та керування транзакцією, викликаючи Commit або Rollback в залежності від успіху виконання.

Метод CommitTransactionAsync також має перевантажені версії для випадку, коли потрібно повертати результат операції.

Метод CommitAsync викликає SaveChangesAsync контексту бази даних, щоб зберегти всі зміни, внесені під час транзакції.

Метод Dispose відповідає за звільнення ресурсів та видалення області сервісів при завершенні роботи з UnitOfWork.

Загалом, UnitOfWork є ключовим компонентом для управління транзакціями та забезпечення атомарності операцій у контексті взаємодії з базою даних.

```
17  public sealed class AuthService : IAuthService
18  {
19      private readonly IUnitOfWork _unitOfWork;
20      private readonly IHashingProvider _hashingProvider;
21      private readonly IDateTimeProvider _dateTimeProvider;
22      private readonly AuthOptions _authOptions;
23      private readonly IJwtTokenVerifier _jwtTokenVerifier;
24
25      public AuthService(
26          IUnitOfWork unitOfWork,
27          IHashingProvider hashingProvider,
28          IDateTimeProvider dateTimeProvider,
29          IOptions<AuthOptions> authOptions,
30          IJwtTokenVerifier jwtTokenVerifier)
31      {
32          _unitOfWork = unitOfWork;
33          _hashingProvider = hashingProvider;
34          _dateTimeProvider = dateTimeProvider;
35          _jwtTokenVerifier = jwtTokenVerifier;
36          _authOptions = authOptions.Value;
37      }
38
39      public async Task<UserCreatedResponse> CreateUser(UserRegisterRequest request)
```

Рис. 3.14 Клас AuthService.cs

AuthService - це службовий клас, який відповідає за реалізацію основних операцій автентифікації та авторизації користувачів у системі.

У конструкторі AuthService отримує залежності, такі як IUnitOfWork, IHashingProvider, IDateTimeProvider, IOptions<AuthOptions>, та IJwtTokenVerifier. Ці залежності використовуються для взаємодії з репозиторіями, провайдерами часу, конфігураційними параметрами та іншими сервісами, необхідними для автентифікації та авторизації.

Метод CreateUser використовується для створення нового користувача, перевіряючи унікальність електронної пошти та додаючи його до бази даних.

Методи GenerateJwtSession та RefreshJwtSession відповідають за генерацію та оновлення токенів доступу та оновлення відповідно. Вони використовують транзакції для забезпечення атомарності операцій та зберігання змін у базі даних.

AuthService також містить допоміжні методи, такі як EnsureRefreshRequestIsValid, який перевіряє валідність та коректність токенів під час оновлення, і IsRefreshTokenExpired, який перевіряє час дії токену оновлення.

Клас взаємодіє з об'єктами користувачів, токенів та репозиторіями, і використовує конфігураційні параметри для генерації та управління токенами. Все це допомагає забезпечити безпеку та надійність механізму автентифікації в додатку.

```
public sealed class ProfileService : IProfileService
{
    private readonly IUnitOfWork _unitOfWork;
    private readonly IObjectsMapper _objectsMapper;
    private readonly IJwtTokenReader _jwtTokenReader;

    public ProfileService(
        IUnitOfWork unitOfWork,
        IObjectsMapper objectsMapper,
        IJwtTokenReader jwtTokenReader)
    {
        _unitOfWork = unitOfWork;
        _objectsMapper = objectsMapper;
        _jwtTokenReader = jwtTokenReader;
    }

    public async Task<CurrentUserProfileResponse> GetCurrentProfile()
    {
        if (!_jwtTokenReader.IsLoggedIn)
        {
            throw new InvalidOperationException("Authorization is required for this operation");
        }

        var currentUserId = _jwtTokenReader.GetUserId();
        var currentUser = await _unitOfWork.Users.TryGet(currentUserId, withTracking: false);
```

Рис. 3.15 Клас ProfileService.cs

ProfileService є класом служби, призначеним для обробки операцій, пов'язаних з отриманням та відображенням інформації про профіль користувача в додатку. У конструкторі він отримує три залежності: IUnitOfWork для взаємодії з репозиторіями даних, IObjectsMapper для відображення об'єктів, та IJwtTokenReader для читання JWT-токена.

Головний метод цього класу - GetCurrentProfile. Під час виклику цього методу, спочатку перевіряється, чи користувач авторизований через JWT-токен. У випадку, якщо користувач не авторизований, генерується виняток типу InvalidOperationException. Якщо ж користувач авторизований, отримується ідентифікатор користувача з токена, та виконується запит до IUnitOfWork для отримання даних профілю користувача. Після отримання цих даних, використовується IObjectsMapper для відображення об'єкта користувача на CurrentUserProfileResponse. Результатом роботи методу є об'єкт відповіді, який містить інформацію про поточний профіль користувача.

У цілому, ProfileService взаємодіє з іншими компонентами додатку, поєднуючи їх для забезпечення функціоналітету отримання та відображення інформації про профіль користувача з урахуванням правил авторизації.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace TrucksProject.Core.Exceptions;

public class CoreException : Exception
{
    protected CoreException(string message, string identifier)
        : base(message)
    {
        Identifier = identifier;
        PropertyErrors = Array.Empty<PropertyErrors>();
    }

    protected CoreException(IEnumerable<PropertyErrors> propertyErrors, string identifier)
        : base("Validation failed.")
    {
        Identifier = identifier;
        PropertyErrors = propertyErrors.ToArray();
    }

    public string Identifier { get; }
    public PropertyErrors[] PropertyErrors { get; }
}
```

Рис. 3.16 Клас CoreException.cs

CoreException є класом винятків у ядрі додатка, який розширює клас Exception. Цей клас призначений для використання у ситуаціях, коли виникають помилки чи виключення пов'язані з основною логікою додатка.

Клас має два конструктори, кожен з яких приймає рядок message та рядок identifier. Перший конструктор використовується, коли виникає помилка без

пов'язаних з нею конкретних помилкових властивостей. Другий конструктор призначений для ситуацій валідації, коли додатково передаються помилкові властивості (propertyErrors). В цьому випадку повідомлення помилки встановлюється на "Validation failed". Identifier - рядок, що ідентифікує тип або джерело помилки. PropertyErrors - масив об'єктів PropertyErrors, який містить інформацію про помилки властивостей.

Є захищене поле PropertyErrors, яке представляє собою масив помилкових властивостей. Це поле ініціалізується пустим масивом у конструкторі, який не приймає помилкових властивостей.

```
0 references
public static class ServiceCollectionExtensions
{
    1 reference
    public static IServiceCollection AddApplicationServices(
        this IServiceCollection services,
        IConfiguration configuration)
    {
        services.AddScoped<IObjectsMapper, ObjectsMapper>();
        services.AddScoped<IHashingProvider, HashingProvider>();
        services.AddScoped<IDateTimeProvider, DateTimeProvider>();
        services.AddScoped<IAuthService, AuthService>();
        services.AddScoped<IProfileService, ProfileService>();

        services.AddScoped<IAuthValidatorsAggregate, AuthValidatorsAggregate>();

        return services;
    }

    1 reference
    public static IServiceCollection AddBroker(this IServiceCollection services)
    {
        services.AddMassTransit(configurator =>
        {
            configurator.AddConsumer(typeof(SampleCommandConsumer));

            configurator.UsingInMemory((context, config) =>
            {
                config.ConfigureEndpoints(context);
```

Рис. 3.17 Клас ServiceCollectionExtensions.cs

ServiceCollectionExtensions - це статичний клас розширень для IServiceCollection, який надає два публічні методи розширення. Перший метод, AddApplicationServices, призначений для додавання різноманітних служб додатка у контейнер залежностей. Він реєструє служби, такі як маппер об'єктів, постачальник хешування, постачальник дати та часу, сервіс автентифікації, сервіс профілю та інші. Кожна служба має життєвий цикл "Scoped", і також додається агрегатор валідаторів для автентифікації.

Другий метод, AddBroker, відповідає за конфігурацію MassTransit для обробки повідомлень в додатку. Він додає консюмера для обробки повідомлень (у цьому випадку SampleCommandConsumer) та використовує провайдер шини повідомлень In-Memory для забезпечення простого тестування та розробки. Крім того, метод конфігурує та реєструє всі необхідні компоненти MassTransit для правильної роботи в контексті додатка. Це спрощує процес конфігурації та реєстрації компонентів у контейнері залежностей та додає базову підтримку для обробки повідомлень через MassTransit.

3.3 Опис методу визначення найбільш вигідних транспортних засобів

Для реалізації цього методу, було використано мову програмування Python та IDE PyCharm. Потрібно було описати математичну модель(задати її), описати умови для розв’язання транспортної задачі, додати обмеження та отримати результат. Перша дія, вводимо змінні, де будуть зберігатися дані про кількість вантажів та транспортних засобів.

```
def find_optimal_transport(cargo_weights, vehicle_capacities, costs):
    # Кількість вантажів та транспортних засобів
    N = len(cargo_weights)
    M = len(vehicle_capacities)
```

Рис. 3.18 Оголошення змінних для зберігання поданих даних

Після цього, задаються параметри моделі. В даному випадку, визначається за допомогою якого вирішувача будуть виконуватись розрахунки. Через те, що присутні обмеження в задачі, необхідно обрати вирішувач для роботи з цілочисельними програмами з обмеженнями, або SCIP (Solving Constraint Integer Programs).

```
# Створення моделі
solver = pywraplp.Solver.CreateSolver('SCIP')
```

Рис. 3.19 Створення відповідної моделі

Далі, створюються бінарні змінні рішення та задається функція, в даному випадку мінімізації вартості.

```
# Створення бінарних змінних рішення
x = [[solver.BoolVar(f'cargo_{i}_{j}') for j in range(M)] for i in range(N)]

# Додавання функції мети (мінімізація вартості)
objective = solver.Objective()
for i in range(N):
    for j in range(M):
        objective.SetCoefficient(x[i][j], costs[i][j])
objective.SetMinimization()
```

Рис. 3.20 Створення бінарних змінних та задання функції мети

Далі, додаються обмеження.

```
# Додавання обмежень
for i in range(N):
    cargo_expr = solver.Sum(x[i][j] for j in range(M))
    solver.Add(cargo_expr == 1)

for j in range(M):
    vehicle_expr = solver.Sum(x[i][j] * cargo_weights[i] for i in range(N))
    solver.Add(vehicle_expr <= vehicle_capacities[j])
```

Рис. 3.21 Задання обмежень

Останній крок – розв’язання самої задачі. Отримання списку транспортних засобів та розрахунок кінцевої суми.

```
# Розв'язання задачі
solver.Solve()

for i in range(N):
    for j in range(M):
        print(f"x_{i}_{j} = {x[i][j].solution_value()}")

# Отримання результатів
selected_vehicles = [j for j in range(M) if any(x[i][j].solution_value() for i in range(N))]

# Розрахунок кінцевої суми
total_cost = sum(costs[i][j] * x[i][j].solution_value() for i in range(N) for j in range(M))

return selected_vehicles, total_cost
```

Рис. 3.22 Розв’язання задачі та отримання результатів

Під час роботи WEB API, дані будуть подаватися безпосередньо через нього, викликом мікросервісу Після виклику такого мікросервісу, з поданням туди необхідних даних, ми отримаємо такий вигляд:

Рис. 3.23 Результат виклику мікросервісу

3.4 Опис бази даних

Одним із ключових елементів при розробці системи є планування складовою бази даних. Дуже важливо на початковому етапі спланувати взаємодію між сутностями. Якщо неправильно спроєктувати взаємодію та наповнення самих сутностей, виправити це буде досить важкою задачею.

В проєкті сутності представлені наступним чином:

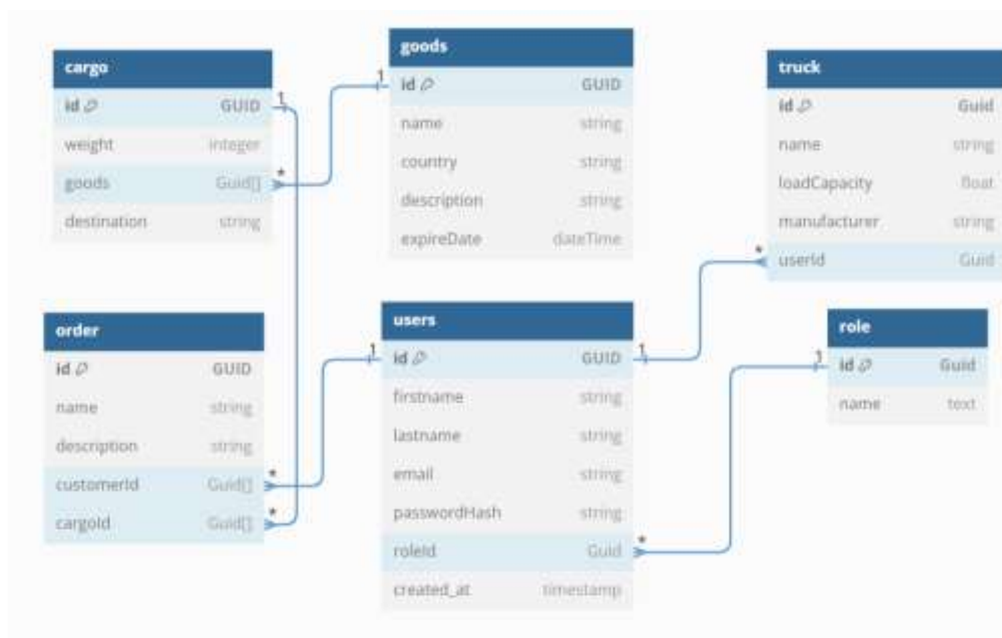


Рис. 3.24 Представлення бази даних сутностей

В системі наявні 6 сутностей, а саме Замовлення, Користувач, Роль, Транспортний засіб, Вантаж та Товар, який знаходиться в Вантажі. В кожній із сутностей обов’язковим атрибутів виступає Id, так як по ньому буде відбуватися пошук тих, чи інших даних. Це не є аксіомою, але частіше за все, це саме Id, так як швидкість пошуку даних таким чином швидша, ніж через будь-який інший атрибут.

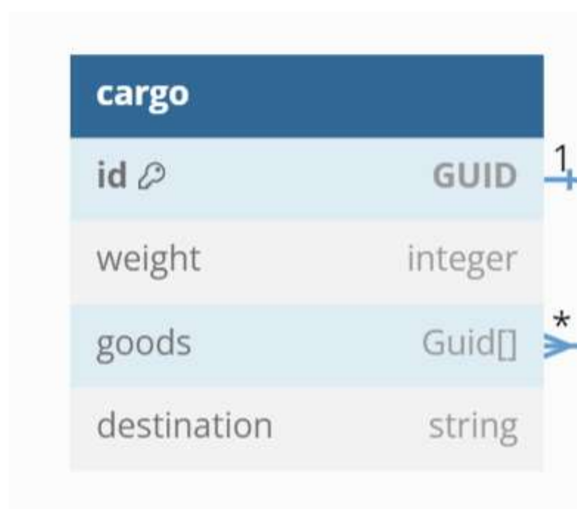


Рис. 3.25 Сутність Cargo(Вантаж)

Дана сутність відповідає за вантаж, який перевозиться. В ньому головним ключем є Id. Присутня також вага вантажу, на яку буде звертати увагу перевізник.

Охарактеризувати сутність можна наступним чином:

- Id (Guid): Унікальний ідентифікатор для кожного вантажу, який використовується як первинний ключ (primary key).
- Weight (integer): Це поле визначає вагу вантажу, виражену цілим числом.
- Goods (Guid[]): Це масив ідентифікаторів (GUID) товарів, які входять до складу даного вантажу. Кожен вантаж може містити декілька товарів.
- Destination (string): Вказує місце призначення вантажу, тобто кінцеву точку, куди вантаж повинен бути доставлений.

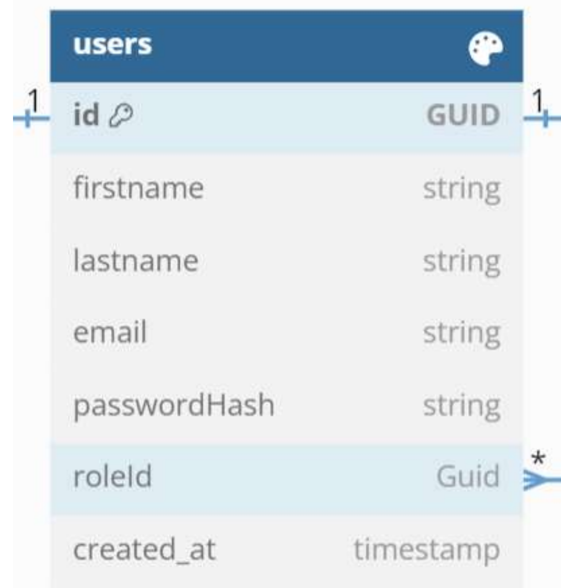


Рис. 3.26 Сутність Users(Користувач)

Сутність "Users" містить основну інформацію про користувачів системи, включаючи їхнє ім'я, прізвище, електронну пошту, хеш пароля, ідентифікатор ролі та дату створення запису.

Охарактеризувати сутність можна наступним чином:

– Id (Guid): Унікальний ідентифікатор для кожного користувача, який використовується як первинний ключ (primary key).
– Firstname (string): Поле, що містить ім'я користувача.
– Lastname (string): Поле, що містить прізвище користувача.
– Email (string): Унікальний ідентифікатор електронної пошти користувача.
– PasswordHash (string): Хеш пароля користувача. Це поле містить захешовану версію пароля з метою безпеки.
– RoleId (Guid): Ідентифікатор ролі, яку має користувач. Посилання на таблицю з ролями.ч’

Рис. 3.27 Сутність Goods(Товар)

Сутність "Goods" містить інформацію про різні товари, зокрема їхню назву, країну походження, опис та дату закінчення терміну придатності.

Охарактеризувати сутність можна наступним чином:

– Id (Guid): Унікальний ідентифікатор для кожного товару, який використовується як первинний ключ (primary key).
– Name (string): Назва товару або товарної позиції.
– Country (string): Країна походження товару.
– Description (string): Опис товару, який може містити додаткову інформацію про товар.

– ExpireDate (dateTime): Дата закінчення терміну придатності товару. Це поле вказує на той момент часу, коли товар стає непридатним для використання.

Рис. 3.28 Сутність Truck(Вантажівка)

Сутність "Truck" містить дані про вантажівки, такі як їхню назву, грузопідйомність, виробника, а також посилання на користувача, якому належить ця вантажівка.

Охарактеризувати сутність можна наступним чином:

– Id (Guid): Унікальний ідентифікатор для кожного вантажівки, використовується як первинний ключ (primary key).
– Name (string): Назва вантажівки або інша ідентифікуюча інформація.
– LoadCapacity (float): Грузопідйомність вантажівки, виражена у вагових одиницях (тонна).
– Manufacturer (string): Інформація про виробника вантажівки.
– UserId (GUID): Унікальний ідентифікатор користувача (власника), який використовується як зовнішній ключ (foreign key) для пов'язання вантажівки з конкретним користувачем.

order
id GUID
name string
description string
customerId Guid[]
cargoId Guid[]

Рис. 3.29 Сутність Order(Замовлення)

Сутність "Order" містить дані про замовлення, такі як назва, опис, користувачі, які замовили його, а також вантажівки або товари, пов'язані з цим замовленням.

Охарактеризувати сутність можна наступним чином:

– Id (Guid): Унікальний ідентифікатор для кожного замовлення, використовується як первинний ключ (primary key).
– Name (string): Назва замовлення або інша ідентифікуюча інформація.
– Description (string): Додатковий опис або характеристика замовлення.
– CustomerId (Guid[]): Масив унікальних ідентифікаторів користувачів, які замовили це замовлення.
– CargoId (Guid[]): Масив унікальних ідентифікаторів вантажівок або товарів, пов'язаних з цим замовленням.

Всі представлені вище сутності також представлені і в програмному коді, але з можливими відмінностями. Всі сутності ми можемо представити в будь якому вигляді та типами даних, для зручності роботи з ними. Як приклад, в програмному коді введено клас EntityBase, який має всього одне поле – Id з типом даних Guid. Цез роблено для зручності при розробці сервісу. Так, як у

кожної Entity є унікальний ідентифікатор, його можна винести в окремий клас, який потім будуть наслідувати ці класи. Це можна виконати і з іншими атрибутами, які є спільними або для всіх Entity, або для більшості з них, щоб не було повторюваності.

```
namespace TrucksProject.Core.Models.Entities;

7 references
public class EntityBase<TId>
{
    2 references
    protected EntityBase(TId id)
    {
        Id = id;
    }

    0 references
    protected EntityBase()
    {
    }

    10 references
    public TId Id { get; }
}
```

Рис. 3.30 Приклад винесення спільних атрибутів в класі EntityBase

3.5 Приклад застосування

Замовник, який виклав своє замовлення для пошуку перевізника, закрив набір перевізників на це замовлення, і хоче отримати список найбільш підходящих транспортних засобів. Для цього, користувач подає Id замовлення(у випадку, якщо буде реалізований Front-end, користувачу нічого вводити самостійно не потрібно, це все виконається автоматично), та відсилає запит. Сервіс, обробивши запит, попередньо провалідувавши його, використовуючи методи валідації, викликає мікросервіс, написаний на мові програмування Python. Сервіс подає необхідні дані на мікросервіс, який оброблює дані та формує відповідь.

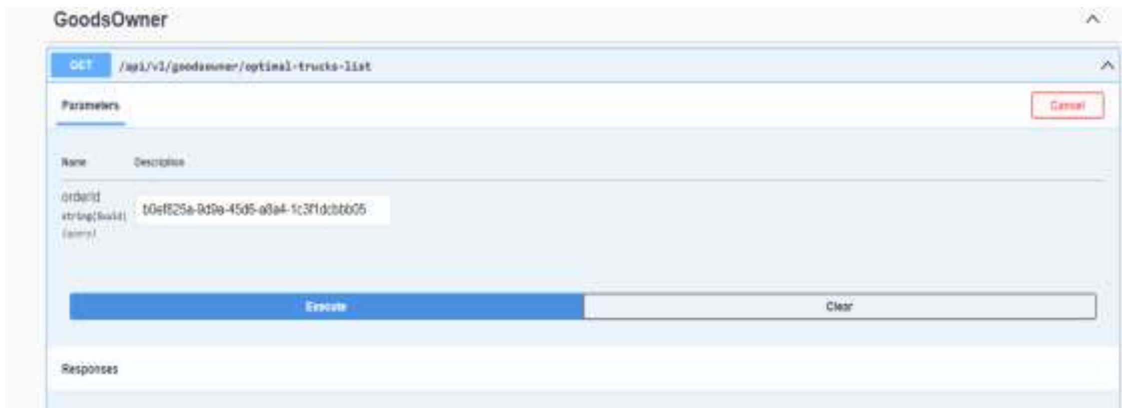


Рис. 3.31 Звернення до WEB API для отримання списку інформації про перевізників

Після отриманої відповіді, замовник може або самостійно обрати необхідного йому перевізника, або довіритись методу та обрати перевізників, транспортні засоби яких було рекомендовано методом. Окрім самих транспортних засобів, метод також і повертає розраховану вартість, яка утворюється при обрахунках.

Рис. 3.32 Отримання результату з боку WEB API

ВИСНОВКИ

У результаті виконання магістерської роботи було розглянуто головні підходи для реалізації автоматизованої генерації ігрової карти для ігор жанру Roguelike та виявлені ключові елементи ігрової карти, що притаманні іграм цього жанру.

Розглянуто такі архітектурні рішення побудови WEB API, як монолітна, сервісна та мікросервісна архітектури. Виконано аналіз особливостей та недоліків цих архітектур, Розглянуто особливості розробки веб-сервісів. Виконано огляд методів та алгоритмів для покращення продуктивності, безпеки та масштабованості WEB API.

Розроблена математична модель, яка мінімізує вартість перевезення вантажу, за допомогою якої було розроблено метод визначення найбільш підходящих транспортних засобів.

Визначені критерії оцінювання вартості перевезення вантажу, за допомогою яких проведено аналіз підбору найбільш підходящих транспортних засобів.

Розроблено програмне забезпечення, що оптимізує логістичний процес підбору перевізника для замовника. Було впроваджено метод визначення найбільш підходящих транспортних засобів за допомогою мікросервісу.

Проведено порівняльну характеристику обробки замовлення. Якщо брати спосіб з оператором, на кожне замовлення витрачається від 30 хвилин і більше, це залежить від людського фактору, кількості необхідної для опрацювання інформації і т.д. WEB API дозволяє скоротити час до 5 сек, з врахуванням затримок на сервері.

ПЕРЕЛІК ПОСИЛАНЬ

1. Carr C., Ramezani C. A. APIs: The (Potential) Digital Connectivity Accelerant For Small and Medium-Sized Importers, Exporters and Their Logistics Providers //Journal of Transportation Law, Logistics and Policy. – 2022. – Т. 89. – №. 1. – С. 18-94.
2. GALLAY, Olivier, et al. A peer-to-peer platform for decentralized logistics. In: Proceedings of the Hamburg International Conference of Logistics (HICL) // epubli, - 2017 - С. 19-34
3. SITHOLE, Beverley; SILVA, Sergio Guedes; KAVELJ, Mirjana. Supply chain optimization: enhancing end-to-end visibility. Procedia engineering // 2016, 159: С. 12-18
4. Meshaal S., Saif A. Microservices and Web-Services: A Review //Peta International Journal of Social Science and Humanity. – 2023. – Т. 1. – №. 1
5. Raj V., Sadam R. Evaluation of SOA-based web services and microservices architecture using complexity metrics //SN Computer Science. – 2021. – Т. 2. – С. 1-10.
6. Kirichek G. et al. Implementation of web system optimization method //CMIS. – 2020. – С. 199-210.
7. Čaušević S. et al. Integration of logistics information systems with electronic sales channels //International Scientific Conference "Science and Traffic Development"-ZIRP 2018. – 2018. – С. 53-62.
8. Müller M. Mobile order-management system: by the example of K-Con logistics service provider. – 2014.
9. Dębicki T. Influence of API interfaces on data exchange and information sharing in the transport and logistics sector //Business Logistics in Modern Management. – 2020.
10. Flores R. et al. Examining Security Risks in Logistics APIs Used by Online Shopping Platforms.
11. Hatvala A. et al. Open innovation opportunities and business benefits of web APIs: a case study of Finnish API providers. – 2016.

12. Tran G. K. Microservice architecture in logistics business. – 2020.
13. Möller F. et al. Data-driven business models in logistics: a taxonomy of optimization and visibility services. – 2020.
14. Omar B., Ballal T. Intelligent wireless web services: context-aware computing in construction-logistics supply chain //ITcon. – 2009. – Т. 14. – №. Specia. – С. 289-308.
15. Delgado Tello E. G. Industry 4.0: Application of advanced services in logistics : дис. – Universitat Politècnica de Catalunya, 2018.
16. Lin X. Z. Unified traceability information system of logistics pallet based on the Internet of Things //Advanced Materials Research. – 2013. – Т. 765. – С. 1181-1185.
17. Shalannanda W. et al. Application for rural internet access services logistics travel duration in Indonesia //2020 6th International Conference on Wireless and Telematics (ICWT). – IEEE, 2020. – С. 1-5.
18. Castro M., Jara A. J., Skarmeta A. Architecture for improving terrestrial logistics based on the web of things //Sensors. – 2012. – Т. 12. – №. 5. – С. 6538-6575.
19. San Jose J. I. et al. Four-layer architecture for product traceability in logistic applications //Big Data and Internet of Things: A Roadmap for Smart Environments. – 2014. – С. 401-423.
20. Kolinski A. et al. Review Of Intelligent Solutions To Optimise Logistics Processes And Improve Efficiency //Bus. Logist. Mod. Manag. – 2021. – Т. 21. – С. 327-349.
21. PAPAZOGLOU, Michael P.; DUBRAY, Jean-jacques. A survey of web service technologies. 2004

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ
(Презентація)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

МАГІСТЕРСЬКА РОБОТА

« ОПТИМІЗАЦІЯ ЛОГІСТИЧНИХ ПРОЦЕСІВ У СФЕРІ ВАНТАЖНИХ ПЕРЕВЕЗЕНЬ ЗА ДОПОМОГОЮ ТЕХНОЛОГІЇ WEB API »

Виконав: студент групи ПДМ-61, Мельник Максим Аркадійович

Керівник: к.т.н., доц., доцент кафедри ІТ Трінтіна Наталія Альбертівна

Київ - 2024

МЕТА, ОБ'ЄКТ, ПРЕДМЕТ ДОСЛІДЖЕННЯ

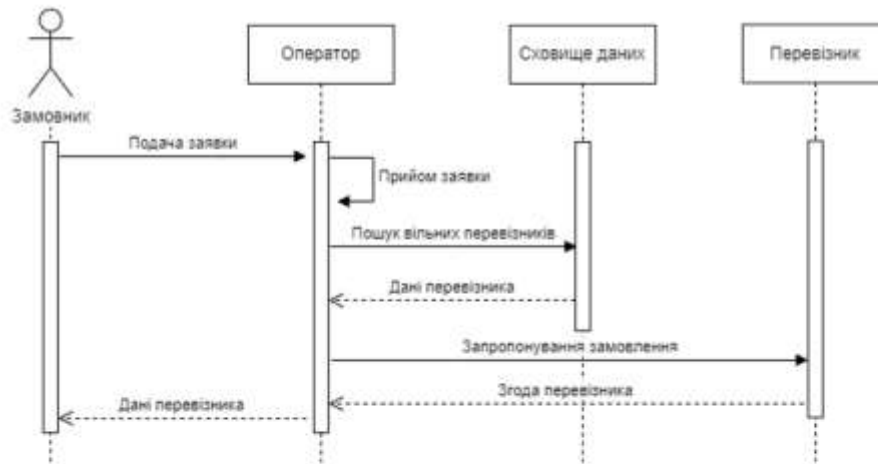
Мета роботи: підвищення якості процесу замовлення перевізника

Об'єкт дослідження: процес замовлення перевізника

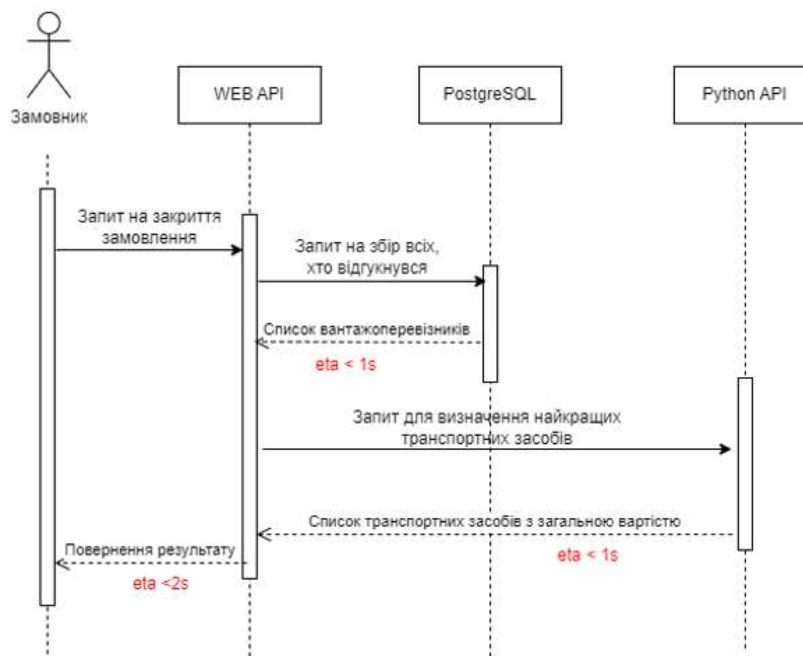
Предмет дослідження: технології WEB API для оптимізації взаємодії між замовником та перевізником.

2

ДІАГРАМА ПОСЛІДОВНОСТЕЙ ДЛЯ ПОДАННЯ ЗАЯВКИ НА ПЕРЕВІЗНИКА ЗАМОВНИКОМ ЧЕРЕЗ ОПЕРАТОРА



ДІАГРАМА ПОСЛІДОВНОСТЕЙ ДЛЯ ВИЗНАЧЕННЯ ЗАМОВНИКОМ НАЙБІЛЬШ ПІДХОДЯЩИХ ТРАНСПОРТНИХ ЗАСОБІВ



ВИЗНАЧЕННЯ КРИТЕРІЮ ЯКОСТІ

Час обробки замовлення: $t_{total} = t_1 + t_2 + t_3 + t_4$

t_1 – час прийому замовлення
t_2 – час пошуку перевізників, які відповідають критеріям замовлення
t_3 – час підтвердження перевізника
t_4 – час передачі інформації замовнику

Точність = $\frac{N_{error}}{N_{total}} * 100\%$, де N_{error} – кількість помилкових запитів
N_{total} – загальна кількість запитів

МАТЕМАТИЧНА МОДЕЛЬ НАЙБІЛЬШ ВИГІДНИХ ТРАНСПОРТНИХ ЗАСОБІВ СЕРЕД ЗАПРОПОНОВАНИХ

Метою методу аналізу визначення найбільш вигідних транспортних засобів є мінімізація вартості перевезення.

$$\sum_{i=1}^{N}\sum_{j=1}^{M} c_{ij} * x_{ij} \to min$$

Обмеження:

$$\begin{cases} \sum_{j=1}^{M} x_{ij} = 1 \\ \sum_{i=1}^{N}\sum_{j=1}^{M} x_{ij} * w_i \le cp_j \end{cases}, \text{де}$$

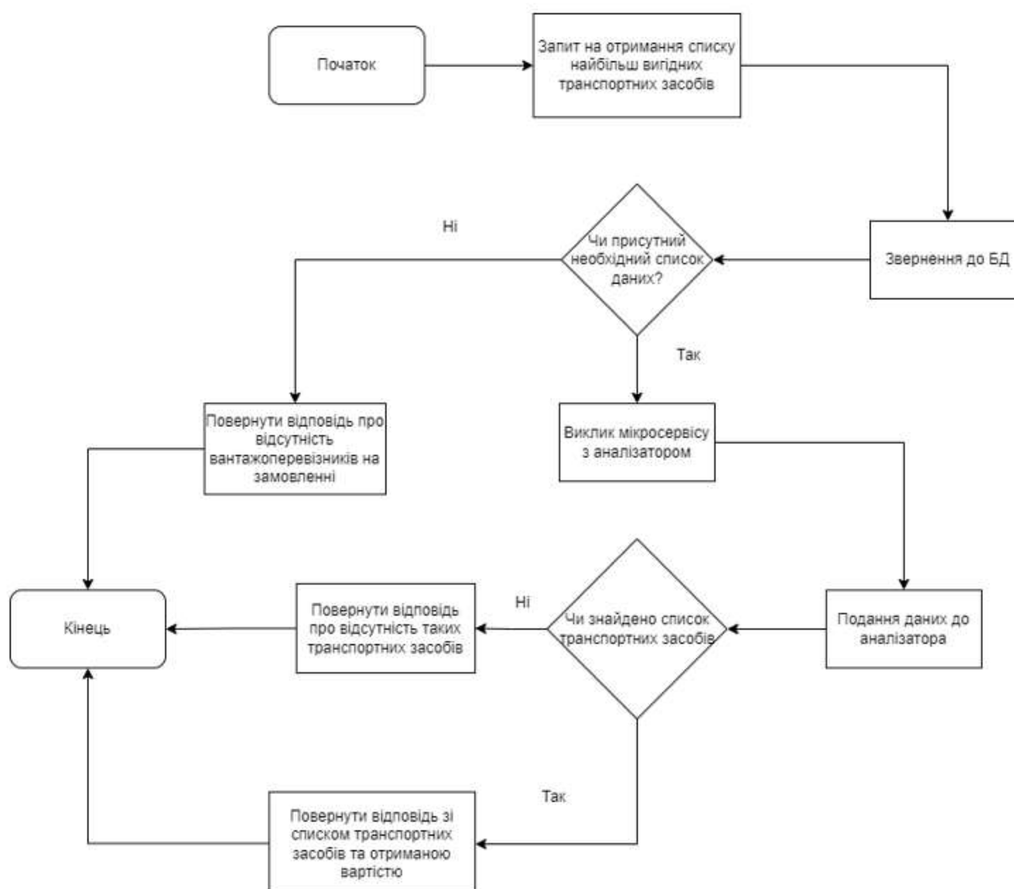
N – кількість вантажів;
M – кількість транспортних засобів;
w_i – вага вантажу i;
cp_j - вантажопідйомність транспортного засобу j;

c_{ij} – вартість перевезення вантажу i транспортним засобом j;
x_{ij} – бінарна змінна, яка дорівнює 1, якщо вантаж i перевозиться транспортним засобом j, і 0 в іншому випадку.

КРИТЕРІЇ ОЦІНЮВАННЯ ВАРТОСТІ ПЕРЕВЕЗЕННЯ ВАНТАЖУ

Витрати на пальне	Ціна палива, яка розраховується на весь проміжок подорожі та включає в себе ціну витраченого палива на кожен пройдений кілометр транспортним засобом.
Витрати на обслуговування	Заміна мастила, необхідні профілактичні роботи та ін.
Вартість перевезення вантажу	Ціна на перевезення вантажу, яка вказана замовником.
Коефіцієнти	Можливі значення коефіцієнтів у випадку різниці регіональних цін.

АЛГОРИТМ ЗАПИТУ НА ЗНАХОДЖЕННЯ ОПТИМАЛЬНИХ ТРАНСПОРТНИХ ЗАСОБІВ



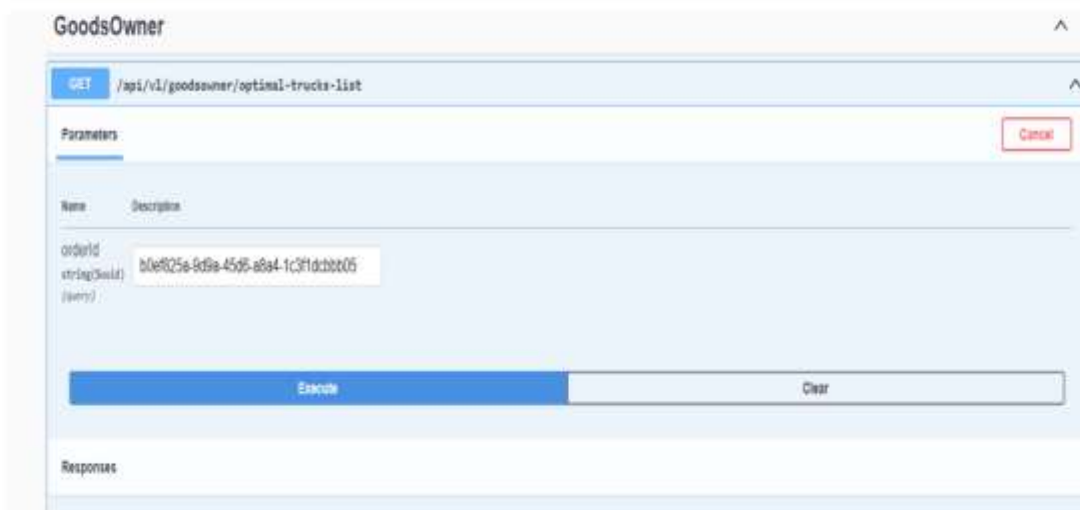
8

ПОРІВНЯЛЬНИЙ АНАЛІЗ ОБРОБКИ ЗАМОВЛЕННЯ КОРИСТУВАЧА

Показники	До запровадження	Після
Кількість замовлень	20	20
Час обробки одного замовлення	Від 30 хв і більше	До 5 сек
Успішно оброблених	16	20
Точність	80%	100%

9

ПРАКТИЧНИЙ РЕЗУЛЬТАТ



“ Звернення до WEB API для отримання списку інформації про перевізників ”

10

ПРАКТИЧНИЙ РЕЗУЛЬТАТ

“ Отримання результату з боку WEB API ”

11

ВИСНОВКИ

1. Визначені критерії оцінювання вартості перевезення вантажу, за допомогою яких проведено аналіз підбору найбільш підходящих транспортних засобів. Додатково визначено критерій якість яка визначається за часом обробки замовлення та точністю.
2. Розроблена математична модель, яка враховує вартість перевезення вантажу, інтегруючи дані з WEB API.
3. Оптимізовано систему управління логістичними процесами за допомогою використання технології WEB API та впровадженню методу підбору найбільш підходящих транспортних засобів.
4. Розроблено WEB API, який реалізує метод підбору найбільш підходящих транспортних засобів.
5. Проведено порівняльну характеристику обробки замовлення. Якщо брати спосіб з оператором, на кожне замовлення витрачається від 30 хвилин і більше, це залежить від людського фактору, кількості необхідної для опрацювання інформації і т.д. WEB API дозволяє скоротити час до 5 сек, з врахуванням затримок на сервері.

12

АПРОБАЦІЯ РОБОТИ

Стаття:

Мельник М.А., Трінтіна Н.А. Вплив технології WEB API на ефективність вантажних перевезень // Телекомунікаційні та інформаційні технології. №1, 2024. (прийнято до друку)

Тези доповідей:

Трінтіна Н.А., Мельник М.А. Паралельне оброблення запитів у веб-сервері: підвищення продуктивності та ефективності веб-сервісів. // V Міжнародна науково-практична конференція молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології» (SoftTech-2023). – подано до друку.

ДЯКУЮ ЗА УВАГУ!